Doctoral theses at NTNU, 2024:106

Felix Schuckert

Opportunities of Insecurity Refactoring for Training and Software Development

Doctoral thesis

NTNU
Norwegian University of Science and Technology
Thesis for the Degree of Philosophiae Doctor
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of Science and Technology

Felix Schuckert

Opportunities of Insecurity Refactoring for Training and Software Development

Thesis for the Degree of Philosophiae Doctor

Gjøvik, May 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of Science and Technology

Thesis for the Degree of Philosophiae Doctor

Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

ISBN 978-82-326-7806-8 (printed ver.)
ISBN 978-82-326-7805-1 (electronic ver.)
ISSN 1503-8181 (printed ver.)
ISSN 2703-8084 (online ver.)

Doctoral theses at NTNU, 2024:106

Printed by NTNU Grafisk senter

Contents

Abstract

Teaching software security is complex and should involve practical exercises. Practical exercises require software artifacts and projects that contain vulnerabilities. The vulnerabilities can be exploited to understand their impact on how different sanitization methods can be bypassed and how they can be mitigated. A frequent challenge is the creation of realistic projects that contain such vulnerabilities. The projects can be created manually with a lot of effort and can only be used once because exploiting the same vulnerabilities repeatedly will not provide a learning effect. The goal of this thesis is to provide a solution to automatically create such learning examples that are realistic and provide permutations that can be used repeatedly for teaching.

As a solution to create learning examples automatically, we invented Insecurity Refactoring. Insecurity Refactoring is a change to the internal structure of a software to inject a vulnerability without changing the observable behavior in a normal use case scenario. Creating realistic vulnerabilities requires characterizing realistic vulnerabilities and identifying how they look like. To solve this challenge, we have reviewed the source code from 150 vulnerabilities that occurred in open-source projects in the categories SQL Injection, Cross Site Scripting and Buffer Overflow. From these vulnerabilities, we have categorized source code patterns of sources, sanitization, context, sinks, and fixes. Those patterns characterize realistic vulnerabilities and are used for Insecurity Refactoring. Additionally, the types of errors from the developers resulting in a vulnerability have been reviewed. Those point out the issues that should be taught to developers and mitigated in the development phase.

Another challenge is creating learning examples that are difficult to detect by static code analysis tools. Those learning examples can also be used to teach developers what source code patterns should be avoided. The previously reviewed vulnerabilities have been scanned with a set of selected commercial and open-source static code analysis tools to identify patterns that produce false positive and false negative results. This insight allows mitigating such

difficult patterns in the development phase to improve the effect of static code analysis. Additionally, these difficult patterns are used in the Insecurity Refactoring approach to create learning examples that cannot be solved by static code analysis tools.

Our method of Insecurity Refactoring has been formalized by using a new defined Adversary Controlled Input Dataflow tree. The formalization allows detecting Possible Injection Paths. Those paths can be transformed into vulnerabilities. All the previously identified source code patterns are used to inject different permutations of vulnerabilities. We developed a tool to realize the formalized method. The tool was tested on open-source projects to check if the approach can inject vulnerabilities. If an open-source project is injectable it does not imply that it is less secure, instead it implies that static code analysis approaches can analyze them to find injection possibilities. The results have indicated that our approach can use 8.1% of the open-source projects found on GitHub to create learning examples.

Projects transformed by our approach have been used as learning examples in two experiments with different groups. The results have shown that the Insecurity Refactoring method does not change the behavior of the program, except when the vulnerability is exploited. Accordingly, the definition of Insecurity Refactoring was confirmed. A survey of the attendees of the experiments has revealed that the transformed projects can be used as learning examples and that the examples are realistic.

Another aspect of this thesis is to improve static code analysis tools. All the identified patterns have been combined to create two static code analysis benchmark data sets. The data sets have been scanned by commercial static code analysis tools. By calculating established static code analysis metrics, the data sets can be used to identify problems of the tools like high false alarm rate, low precision, low recall, etc. Additionally, we have provided a solution to identify patterns that the tools do not cover. The generation process has been discussed in an interview with experts from Software Assurance Metrics And Tool Evaluation (SAMATE) at the National Institute of Standards and Technology (NIST). They have approved that the generation process is solid. The two generated data sets are being hosted as an official Software Assurance Reference Dataset (SARD). This allows all developers of static code analysis tools to test their tools and improve it based on our research.

Acknowledgment

I would like to thank my supervisors, Prof. Hanno Langweg and Prof. Basel Katt for all their patience and help with this PhD. Their knowledge and experience encouraged me all the time of my academic research and daily life. I am thankful that the work has been financially supported by the Auerbach Foundation and HTWG Konstanz. I would also like to thank Max Hildner who helped to process many of the initial code reviews. Additionally, I am thankful that Matthias Wegner, Julian Weißgerber and Sandra Zinsmaier helped me by proofreading. Finally, I would like to express my gratitude to my parents, my wife, and my children. Without their understanding and encouragement in the past few years, it would have been impossible for me to complete my study.

Chapter 1

Introduction

Software development is a complex process and you cannot totally prevent the occurrence of security issues. There are forms of effective formal methods to ensure correctness and some security properties but they are not efficient as of today. Static code analyzer tools try to detect security issues in source code. The results from Goseva-Popstojanova and Perhinschi [40] show that the overall highest detection rate of known security vulnerabilities is 59%. If developers have to fix possible vulnerabilities, they will require software security skills to distinguish between an incorrect report (false positive) or correct report (true positive). The false positive rate remains a problem because developers have to manually review each report. Especially then, the fixing process requires software security skills as well. Depending on the software security skills, developers who try to fix vulnerabilities could create new security issues or just hide the existing one from static code analyzer tools. Accordingly, the involved review process by using static code analyzer tools depends on the software security skills.

The security development life cycle involves software security in the software development process. It includes different steps to prevent security issues already in the development phase. One important aspect is that all developers require security training to prevent security issues. It is recommended to refresh the security skills by frequently participating in training events [41]. Even the usage of static code analysis tools requires software security knowledge to determine a vulnerability report is actually a vulnerability. In addition, the static code analysis tools usually don't provide a solution. Accordingly, the developer also needs software security knowledge to fix the reported vulnerability. A typical approach to teaching software security is hands-on training/labs and it involves working with vulnerable projects. By exploiting a vulnerability, one can learn the effect and impact

of such vulnerabilities. Vulnerabilities can be patched by developers to learn different prevention techniques. A common problem in this context is the creation of these vulnerabilities for realistic learning exercises. Currently, there exist many different projects like WebGoat [8], Juliet [3], et cetera. These projects contain security vulnerabilities based on the most prevalent security issues. The projects are created manually and the exercises can be used only once. It is not useful for repetitive learning as recommended by the software security development life cycle. Furthermore, other solutions exist that automatically generate artificial vulnerabilities [74]. Current solutions are either manually created or automatically created but on the artificial side. The goal is to create vulnerabilities automatically based on vulnerabilities from real software projects.

If developers require training frequently, the exercises should be different each time. Some security vulnerabilities occur in unusual scenarios. For example, the Common Vulnerabilities and Exposures (CVE) 2001-1471 did have an OS injection issue via an invalid language file [2]. Developers have to learn that such issues can occur from all kinds of sources. Having a simple way of generating different exercises based on common security issues will help to teach developers software security skills and raise awareness. The different exercises should be as real as possible to teach how the vulnerabilities occur in software projects.

The benign usage of Insecurity refactoring as learning examples is helpful. In contrast, automatic vulnerability injections can be used harmfully. Accordingly, insecurity refactoring could be used harmfully. A recent attack on the PHP git repository [17] tried to introduce a backdoor into PHP that was falsely flagged as fixing typing errors. This might be a specially crafted backdoor or might have been an automatically created backdoor. A possible attack scenario by using insecurity refactoring is by automatically generating code changes that will be sent to different git repositories. Because it is an automatic approach, the amount of such requests can be very high. If only a small percentage of the requests are accepted, it will be enough for attackers to introduce harmful code changes. Another attack scenario is that an attacker has a short amount of time accessing a code base. A manually crafted backdoor would not be possible in a short time frame. In this case, an automatic approach could be used to inject a backdoor in the code base. It is important to know the limitations of insecurity refactoring, to assess what attackers can do and what not.

As such attack attempts by using automated injected vulnerabilities are critical, it is important to know if these injected vulnerabilities and backdoors can be detected by modern tools. If the tools can detect all

the injected vulnerabilities, such attack attempts will be solved by using these tools. Nevertheless, tools that cannot detect such automated injected vulnerabilities cannot prevent such attacks. Accordingly, these tools should be improved that the impact of automated injected vulnerabilities is as low as possible. Categorized source code patterns help to specify where the different tools have problems. This helps the developers of such tools to improve their tools based on those patterns.

Chapter 2

Research Questions

The primary idea of this work is to create a novel approach called Insecurity Refactoring. It uses static code analysis and refactoring approaches to inject vulnerabilities in existing projects. We define Insecurity Refactoring as follows: *Insecurity refactoring is a change to the internal structure of software to inject a vulnerability without changing the observable behavior in a normal use case scenario. If the injected vulnerability is exploited, the observable behavior will change.* Accordingly, Insecurity Refactoring requires maintaining the normal usage of the program. For example, if a number field is used. The normal usage will be that numbers are inserted and the program should still run as expected. In contrast, insecurity refactored vulnerabilities behaves differently if the user inserts unexpected inputs (e.g., an apostrophe). The exploit scenario is intended as the goal is to inject vulnerabilities. Nevertheless, the process tries to minimize the potential of a crash, but it cannot ensure that it will not happen for unintended inputs. The observable behavior means that the user does not see a difference in the running program as long as numbers are inserted in the number field. In the background, the insecurity refactored code might perform a bit different because the code has been changed, but the output of the program will be the same.

The goal is to use these injected vulnerabilities as learning examples. This requires that the vulnerabilities are as realistic as possible. In addition, it should be possible to inject vulnerabilities that are not detectable by static code analysis tools. This can be used as more advanced learning examples or can be used to teach developers what source code patterns are difficult for static code analysis tools. Depending on what learning examples are required, such patterns can be included or not.

As this thesis is based on vulnerabilities from open-source projects, we

had to decide what programming language and what vulnerabilities will be reviewed. Based on the vulnerabilities we had access to, the focus relies on vulnerabilities of the categories: SQL Injection, Cross Site Scripting and Buffer Overflow. The following section describes the identified scientific questions for Insecurity Refactoring.

2.1 Q1: What are source code patterns that can classify existing security vulnerabilities?

Insecurity Refactoring should be as realistic as possible. Accordingly, to create learning examples matching that requirement it is important to identify how real vulnerabilities look like. Vulnerabilities are already classified by Open Worldwide Application Security Project (OWASP) [10], Common Weakness Enumeration (CWE) [9], et cetera. The generation process of security vulnerabilities requires a more thorough analysis and classification based on the source code.

Is it possible to identify and categorize source code patterns that are found in source code from open-source projects related to injection related CVE reports (SQL Injection, Cross Site Scripting and Buffer Overflow) in PHP projects? The source code patterns are considered as a developer would see them. Accordingly, common patterns like sources, sinks, and sanitization patterns are identified and categorized. Are there any special patterns that software developers have to know to prevent such vulnerabilities? What insufficient sanitization methods can be found in CVE report related source code?

2.2 Q2: What are the limitations and problems of static code analysis tools?

Static code analysis tools have to be evaluated to find out why they do not detect certain types of security issues. The motivation here is to create vulnerabilities that cannot simply be detected by static code analysis tools. Accordingly, the problems or limitations have to be identified. To clarify, this work uses the terms problematic and difficult for static code analysis tools. A problematic or difficult source code pattern means that static code analysis tools cannot analyze it correctly. For example, a tool might not track the data flow if a difficult source code pattern is reached. All of those tools are different, a difficult source code pattern can be difficult for one tool

but does not have to be difficult for all tools. Both patterns, problematic and difficult are used, and there is no difference between the two.

Are there source code patterns that static code analysis tools are unable to detect if it is contained in a vulnerability? Are there sanitization functions that are found in real projects that the static code analysis tools do not detect? Are there insufficient sanitization functions that are found in real projects that the static code analysis tool detect as sufficient? Can detectable security vulnerabilities be modified by an introduced difficult pattern that prevents static code analysis tools from detecting it?

2.3 Q3: How can static code analysis and refactoring approaches be used to inject vulnerabilities?

The main goal of this thesis is to create learning examples with an automated approach. Accordingly, it is important to inject vulnerabilities that look like a developer has written them accidentally.

The main question of this work is, how can Insecurity Refactoring be implemented by using static code analysis and refactoring approaches? How can source code be modified to create security vulnerabilities without modifying the normal usage behavior of the program? How can static code analysis methods be used to modify source code? Can the previously identified patterns be used to create realistic vulnerabilities? Can the previously identified problematic static code analysis patterns be inserted to the injected vulnerabilities?

2.4 Q4: How can training materials derived by insecurity refactoring be used in training software security?

A goal of this research is to improve learning methods for software developers. The following questions are used to evaluate the Insecurity Refactoring approach. Does the implemented Insecurity Refactoring comply with the definition and is the external behavior in normal usage not changing? Are the injected vulnerabilities exploitable? Can the injected vulnerabilities be used to train software security skills? Are the injected vulnerabilities realistic, or do they feel artificial like other automatic generated exercises?

2.5 Q5: How can source code patterns be used to create test suites to improve static code analysis tools?

As insecurity refactoring can also be used harmfully, it is important to enable modern tools to detect such vulnerabilities. How can source code patterns created from newly discovered vulnerabilities be used to create a test suite for SCA tools? Can the difficult source code patterns for SCA tools be included in the test suite? How can the decision of a test case being vulnerable or safe be solved, involving the different contexts and sanitization methods? Is such a test suite based on the identified patterns useful to benchmark static code analysis tools?

Chapter 3

Background

In this section, the relevant works that influenced and provide the background for this work. This thesis is based on classifying vulnerabilities and static code analysis models. Accordingly, the Common Weakness Enumeration (CWE) [9] vulnerability classification and the basic static code analysis models are the background.

3.1 Classification of vulnerabilities

The Common Weakness Enumeration (CWE) [9] combines common software and hardware weakness types. They define weaknesses that include flaws, faults, bugs and other errors that if left unaddressed could result in systems, networks, or hardware being vulnerable to attacks. The goal of the classification is to define a common language for vulnerabilities. The common language can be used to teach software developers to prevent these issues at the development phase. The classification itself is split into many different types (Class, Base, Variant, ...). The different categories have relationships to each other. For example, the class category 'CWE-119:Improper Restriction of Operations within the Bounds of Memory Buffer' is the parent of the base category 'CWE-120: Classical Buffer Overflow'. Additionally, a CWE top 25 is regularly published that contains the top vulnerabilities categories that occurred in a recent time frame. The scoring is based on how many CVE reports have been mapped to the CWE categories. For the scoring, only CVE reports are used that are mapped to CWE categories by the National Vulnerability Database (NVD). The methodology to calculate the scoring

Rank	ID	Name	NVD Count	Avg CVSS	Score
1	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	3788	5.8	46.82
2	CWE-787	Out-of-bounds Write	2225	8.31	46.17
3	CWE-20	Improper Input Validation	1910	7.35	33.47
4	CWE-125	Out-of-bounds Read	1578	7.13	26.5
5	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	1189	8.08	23.73
6	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	901	8.98	20.69
7	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	1467	6.01	19.16
8	CWE-416	Use After Free	918	8.26	18.87
9	CWE-352	Cross-Site Request Forgery (CSRF)	866	8.08	17.29
10	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	767	8.52	16.44
11	CWE-190	Integer Overflow or Wraparound	846	7.7	15.81
12	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	792	7.27	13.67
13	CWE-476	NULL Pointer Dereference	529	6.83	8.35
14	CWE-287	Improper Authentication	412	8.05	8.17
15	CWE-434	Unrestricted Upload of File with Dangerous Type	346	8.5	7.38
16	CWE-732	Incorrect Permission Assignment for Critical Resource	426	6.99	6.95
17	CWE-94	Improper Control of Generation of Code ('Code Injection')	295	8.74	6.53
18	CWE-522	Insufficiently Protected Credentials	283	7.92	5.49
19	CWE-611	Improper Restriction of XML External Entity Reference	277	7.88	5.33
20	CWE-798	Use of Hard-coded Credentials	234	8.76	5.19
21	CWE-502	Deserialization of Untrusted Data	217	8.93	4.93
22	CWE-269	Improper Privilege Management	278	7.36	4.87
23	CWE-400	Uncontrolled Resource Consumption	249	7.09	4.14
24	CWE-306	Missing Authentication for Critical Function	193	8.1	3.85
25	CWE-862	Missing Authorization	236	6.9	3.77

Table 3.1: CWE top 25 - 2020 [9].

requires a frequency of each CWE category c_x [9]:

$$F = \{count(c_X \in NVD \text{ for each } c_X \text{ in } NVD\}$$
$$Fr(c_x) = \frac{count(c_x \in NVD) - min(F)}{max(F) - min(F)}$$

It provides a frequency based on all other mapped CVE reports. The scoring also includes the severity of each CVE report. The severity for each CWE category c_x is calculated as follows:

$$CVSS_x = \{\text{CVSS scoring for each CVE report mapped to } c_x\}$$
$$Sv(c_x) = \frac{average(CVSS_x) - min(CVSS_x)}{max(CVSS_x) - min(CVSS_x)}$$

A combination of the frequency and the severity allows calculating the final scoring:

$$Score(c_x) = Fr(c_x) * Sv(c_x) * 100$$

Table 3.1 shows the scoring that has been published in 2020. The time frame has been the recent two years. Related to the contribution of this thesis, the related categories CWE-79 *Cross Site Scripting* at rank 1, CWE-119 *Buffer Overflow* at rank 5 and CWE-89 *SQL Injection* at rank 6 are still top scoring vulnerabilities. Each of these categories are also split into CWE sub categories that represent the errors in more detail.

Another important classification of top security issues is from the Open Web Application Security Project (OWASP) [15]. The focus relies on web

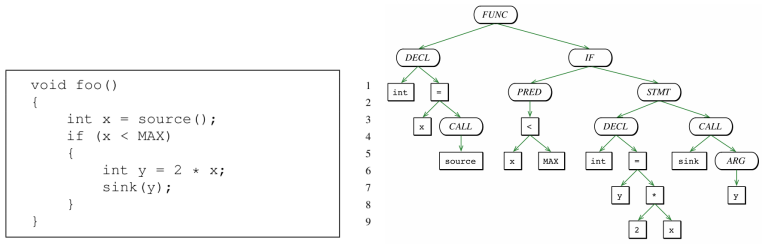
Rank	Name
1	Broken Access Control
2	Cryptographic Failures
3	Injection
4	Insecure Design
5	Security Misconfiguration
6	Vulnerable and Outdated Components
7	Identification and and Authentication Failures
8	Software and Data Integrity Failures
9	Security Logging and Monitoring Failures
10	Server-Side Request Forgery

Table 3.2: OWASP top 10 - 2021 [16].

applications to reduce the occurrence of potential vulnerability types. They use a hybrid methodology to get data for the top 10 list. The three primary sources of data are Human assisted Tooling (HaT), Tool assisted Human (TaH) and raw tooling. Raw tooling and HaT are producing a lot of data because the tools can find many vulnerabilities. Especially, vulnerabilities with the same systematic that occur in multiple locations produce many results. In contrast, TaH has a lower frequency because vulnerabilities that are systematic will only occur once. The top ten use 8 ranks that are from the hybrid methodology, and two ranks are determined by surveys. The surveys are sent to companies asking what web security issues have occurred. The initial top ten list is then published and can be reviewed by the public. After a consensus is reached, the top ten list will be released. Table 3.2 shows the OWASP top ten of 2021. Injection includes SQL Injection that was on rank 1 in 2017 moved to rank 3 in 2021. Cross Site Scripting was at rank 7 and moved to rank 8 in a new category named Software and Data Integrity Failures.

3.2 Static code analysis models

Static code analysis tools are often used to detect software vulnerabilities. For this work, static code analysis models are used to find source code parts that can be used to inject the vulnerability. The static code analysis model that has been used is a combination of the Abstract Syntax Tree, Control Flow Graph and the Program Dependence Graph. Accordingly, it is important to understand the fundamental static code analysis models. The specific model (Code Property Graph) that has been used is then explained in detail in chapter 4. The common static code analysis approach is to create different analysis models that will be used to find bugs, vulnerabilities, and code



(a) Source code sample (b) Abstract Syntax Tree.

Figure 3.1: Source code and corresponding Abstract Syntax Tree [78].

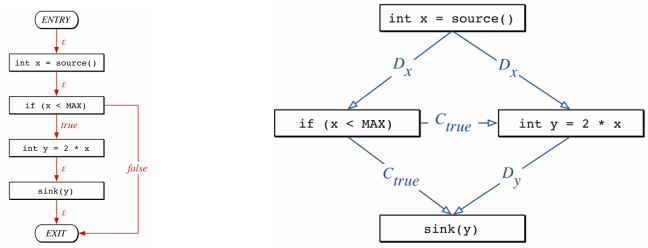
quality issues. This section describes the basic models and one advanced model which is the Code Property Graph that has been used in our thesis.

3.2.1 Abstract Syntax Tree

The basic static code analysis model is the Abstract Syntax Tree (AST). It is an abstract tree representation of the source code. An Interior node represents an operator, and the children represent the operands of the operator [76]. For an addition $2 + 3$, the operator would be $+$, and the children in the AST would be 2 and 3 of the operator. The AST is a simple syntax tree that does some abstraction of the source code. For example, the *echo* statement and $<?=$ are both represented as an echo statement because both return output to the client. The abstraction makes it easier to analyze the tree. Figure 3.1 shows for a source code sample the corresponding AST. The function call *sink* has a function *CALL* node as operator and the operands of that are the function name *sink* and an argument *ARG*. An AST has no specification on how it is constructed. Depending on the implementation, the AST can look thoroughly different. The example in figure 3.1b shows the AST for the source code.

3.2.2 Control Flow Graph

The Control Flow Graph (CFG) is used by compilers and can also be used to analyze source code [23]. The concept is that the source code is split into code blocks. A code block is a list of statements that will be sequentially executed. Accordingly, it has a one start statement and always ends in the same end statement without the possibility that any instruction is skipped [30]. There cannot be any decisions like *if* statements inside a code block. The code blocks will be connected by directed edges to show in which order the code blocks are executed. A backwards edge shows a loop in the source code. For



(a) Control Flow Graph. (b) Program Dependence Graph.

Figure 3.2: Code representation of the program flow and the dependencies [78].

example, an *if* statement is a split in the CFG. One edge is connected to code block, if the *if* statement is *true* and another edge is connected to the code block that is executed when the *if* statements returns *false*. The CFG is defined as a directed graph $G = (B, E)$ where B is the set of blocks and E is the set of directed edges [23].

There are different construction mechanisms for a CFG. For example, it can be generated from binary code [56] [34] [57]. Compilers usually create a CFG based on the AST. Figure 3.2a shows the CFG for the source code sample. A code block is represented by statements that are connected with the label ϵ. Otherwise, it has either *true* or $false$ as label based on the decision outcome. That CFG is constructed by searching for all structured control statements (*if*, *while*, ...) first and creating a preliminary Control Flow Graph. As a next step, the unstructured control statements (*goto*, *break*,...) are used to correct the preliminary Control Flow Graph.

3.2.3 Program Dependence Graph

Ferrante et al. [38] invented the Program Dependence Graph (PDG). The basic concept of a PDG uses statements and predicates as nodes. The PDF is a combination of the Data Dependence Graph and Control Dependence Graph into one graph. The Data Dependence Graph shows for each statement the dependency to other statements. Figure 3.3a shows a code sample where statement S2 depends on the statement S1 because of the variable A that is assigned to statement S1 and used in statement S2. The data dependencies have to preserve the semantic of the program, the statements cannot be reserved. The creation of the Data Dependency Graph can be solved by solving the *reaching definition* [76]. For example, the CFG can be used to solve the *reaching definition* problem.

```
1 A = B * C        (S1)
2 D = A * E + 1    (S2)
```

(a) Data dependency

```
1 if (A) then      (S1)
2    B = C * D     (S2)
3 endif
```

(b) Control dependency

Figure 3.3: Source code samples to show the data dependency and control dependency [38].

The second part of the PDG is the Control Dependence Graph. If a statement depends on the outcome of a predicate, an edge will be created pointing from the predicate to the statement. Figure 3.3b shows a code sample, where the statement S2 depends on the predicate S1. The edge has a label, if the result of the predicate has to be true or false. The Control Dependence Graph can be constructed by using the Control Flow Graph. A *post-dominator tree* [76] can be calculated from the CFG. Then the tree can be used to generate the Control Dependence Graph. A combination of both graphs creates the Program Dependence Graph. The PDG was initially used for program slicing [77]. Figure 3.3b shows the PDG that is used in the Code Property Graph. This allows to easily analyze that the sink depends on the statement where the variable y is being assigned and the predicate of the *if* statement.

Chapter 4

Related Work

This section provides an overview of related work that has influenced this work. Related work that had high influence on this thesis are described in detail.

4.1 Classification of vulnerabilities

Vulnerabilities are classified in many ways. Early classifications were classifying the vulnerabilities itself, e.g., [26, 46, 25]. A well-known classification of vulnerabilities are the Common Weakness Enumeration (CWE) [9] and OWASP top 10 [15]. There is research that defines taxonomies of vulnerabilities also based on mining open-source projects and other resources [71, 71, 28, 73]. Those also have the same top categories like sinks, sanitization, source, etc. Medeiros et al. [53] manual reviewed source to classify sanitization methods. Li et al. [48] classified different vulnerabilities characteristics.

Lerthathairat and Prompoon [47] classify source code into bad, ambiguous and clean. The first step in their approach is to classify source code into either being bad smell, ambiguous or clean code by using software metrics. For each category, different specifications on the metrics are set to define if a source code part is categorized as a corresponding category. For example, the metric lines of code (NLOC) is a specification to check for the category *long method*. The specification state that a function is either clean (x<=NLOC), ambiguous (20<x<60) or bad smell (x>=60). For bad smell and ambiguous source code, they use a fuzzy logic method to transform the source code into a clean state. For the different bad smell categories, different refactoring approaches for mitigation are listed. The fuzzy logic method is used to refactor the project until the project is classified as clean code. Accordingly,

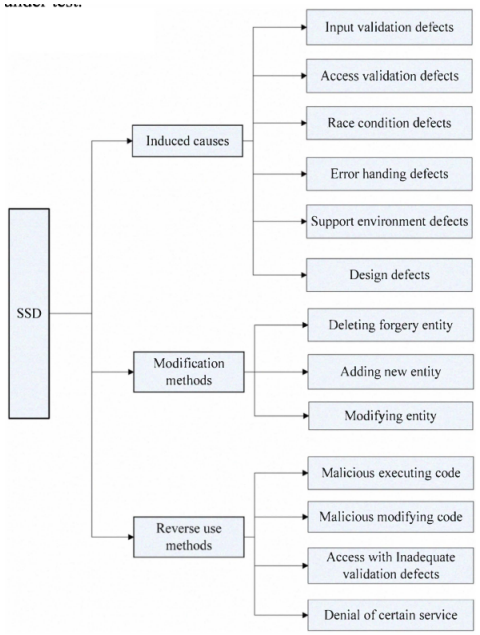


Figure 4.1: Software security defect taxonomy. [43]

the fuzzy logic method tries to get each step closer to clean code metrics.

Hui et al. [43] developed a taxonomy of software security defects (SSD). A software security defect is an error that usually a developer did at the development stage. A SSD can create a software vulnerability. They reviewed vulnerabilities and threat databases and interviewed practitioners in the software security testing field to determine the taxonomy. Figure 4.1 shows the taxonomy. The *Induced causes* are representing the error at the development stage. The six sub categories represent what error has been made. For example, if the developers forgot to add an input validation check on user provided data. The *Modification methods* classify what an attack can do with data entities. The *Reverse use methods* show how the SSD can be detected with a testing strategy. Additionally, they mapped a top 10 list of dangerous programming errors to the categories. They also published a case study where they provide a SSD-based security testing methodology [44].

Shar and Tan [71] [72] also define a taxonomy for Cross Site Scripting and SQL Injection vulnerabilities. Figure 4.2 shows the categories. There are only two categories for sinks that actually represent the following vulnerability types: Cross Site Scripting (*HTML*) and SQL Injection (*SQL*). There are more specific categories for source and sanitization. The categories for sources

are remotely (*Client*), from a *Database* or uninitialized variables (*Uninit*). Sanitization function are split into official sanitization functions (*sanitization*) that are provided by database driver or stem from the official programming language. Another sanitization category is *Encoding* functions that encode differently based on the function. *Encryption* and *Numeric-conversion* are the sanitization functions that return a value which in normal certain stances cannot create any issues. More problematic are the *Replacement* and *Regex-replacement* functions because they depend on regular expression to determine the sanitization functionality. In the *Remaining* category, all the remaining sanitization functions are found. In their research, they use the taxonomy to detect vulnerabilities with machine learning. The approach is to use the Control Flow Graph (CFG) to create a Data Dependence Graph (DDG) models to count the occurrence of each sink category. The models will be explained later. The amount of each pattern is then used as input for machine learning. They used their approach on open-source projects to collect data that is used for evaluation. The results show positive results by having a high true positive rate and a low false positive rate.

The authors saw a problem at the sanitization functions that can either be sufficient or not. In [73], they continued their work by providing a hybrid approach to solve that issue instead of a static approach. Their approach starts with a static code analysis by using the Data Dependence Graph. If it finds a critical function, it continues from there with dynamic testing. The dynamic test uses a set T of critical inputs. That set is filled with critical inputs which are extracted from cheat sheets that are used to bypass XSS and SQLi sanitization functions. The dynamic part tests each critical input $t \in T$ to see if the outputs changes in a specific manner. Based on the output, they define dynamic attributes as seen in table 4.1. They added more attributes to their previous taxonomy, e.g., functions that return a *Boolean*. The further approach uses the found patterns as attributes for machine learning. It can resolve the issue of a sanitization function being sufficient by using the dynamic approach. They used open-source machine learning projects and they looked into the difference between supervised and unsupervised machine learning algorithms for their approach.

Compared to our results, we did find categories that are based on the source code. Our contributions are source code patterns from the view of a developer instead of the view of the vulnerabilities. This helps developers to understand what to mitigate in their source code. In addition, the manual review allowed to extract special cases. Our taxonomy has in mind to be used for code generation. Accordingly, it is source code pattern focused instead of providing a broad vulnerability taxonomy.

Attribute ID	Attribute Name	Description
Static attributes		
1	Client	The number of nodes that access data from HTTP request parameters
2	File	The number of nodes that access data from files
3	Database	The number of nodes that access data from database
4	Text-database	Boolean value 'TRUE' if there is any text-based accessed from database; 'FALSE' otherwise
5	Other-database	Boolean value 'TRUE' if there is any data except text-based data accessed from database; 'FALSE' otherwise
6	Session	The number of nodes that access data from persistent data objects
7	Uninit	The number of nodes that reference un-initialized program variable
8	SQLI-sanitization	The number of nodes that apply standard sanitization functions for preventing SQLI issues
9	XSS-sanitization	The number of nodes that apply standard sanitization functions for preventing XSS issues
10	Numeric-casting	The number of nodes that type-cast data into a numeric type data
11	Numeric-type-check	The number of nodes that perform numeric data type check
12	Encoding	The number of nodes that encode data into a certain format
13	Un-taint	The number of nodes that return predefined information or information not influenced by external users
14	Boolean	The number of nodes which invoke functions that return Boolean value
15	Propagate	The number of nodes that propagate partial or complete value of an input
Dynamic attributes		
16	Numeric	The number of nodes which invoke functions that return only numeric, mathematic, or dash characters
17	LimitLength	The number of nodes that invoke string-length limiting functions
18	URL	The number of nodes that invoke path-filtering functions
19	EventHandler	The number of nodes that invoke event-handler filtering functions
20	HTMLTag	The number of nodes that invoke HTML-tag filtering functions
21	Delimiter	The number of nodes that invoke delimiter filtering functions
22	AlternateEncode	The number of nodes that invoke alternate-character-encoding filtering functions
Target attribute		
23	Vulnerable?	Indicates a class label - Vulnerable or Not-Vulnerable

Table 4.1: Static-Dynamic Hybrid Attributes. [73]

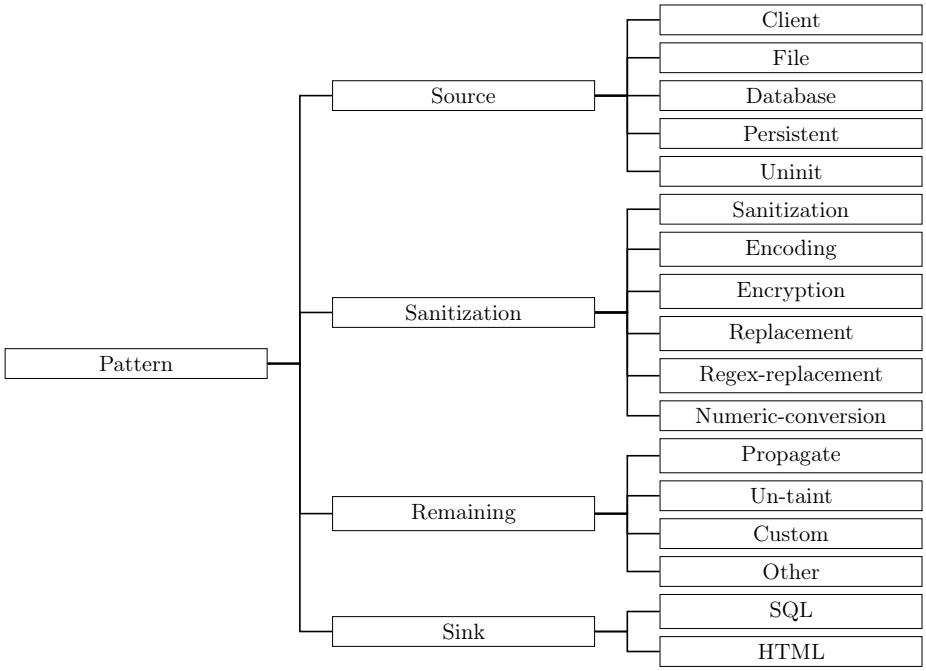


Figure 4.2: Taxonomy that is used as machine learning attributes [72].

4.2 Static code analysis evaluation

The research about how static code analysis tools perform regarding detection rate, false positive rate, etc. is used to compare the tools, e.g., [80, 22, 58, 59]. Even the performance of penetration testing tools against static code analysis tools are evaluated [24].

Goseva-Popstojanova and Perhinschi [40] evaluated three commercial static code analysis tools. Foremost, they evaluated what static code analysis tools are suitable for their research. They started with 22 static code analysis tools and all projects got excluded that either: Have no support for Java/C++, cannot detect security issues, cannot handle a large code base or are distributed as software as a service. No open-source static code analysis tools were used because they perform overall worse than commercial tools [36]. Three commercial tools have passed the evaluation criteria. One tool has been classified as *program verification & property checking*, the other one falls into the category *bug finding* and the last one is aimed for *security review*.

For the first evaluation, a data set from Juliet [3] has been used. The Juliet test suite contains many vulnerabilities related to different CWE categories. To get a data set where the results from each static code analysis tool can be compared to each other, only CWE categories have been selected that are officially supported by all the three static code analysis tools. Then, they used a subset of test cases from the Juliet test suite that are related to these CWE categories as a data set. All chosen data sets have been scanned by all three tools and the following metrics have been calculated:

- Accuracy $Acc = \frac{TN+TP}{TN+FN+FP+TP}$
- Recall $R = \frac{TP}{FN+TP}$
- False alarm rate $F = \frac{FP}{TN+FP}$
- G-Score $G = \frac{2R(1-F)}{R+1-F}$

The G-Score is a harmonic mean of R_i and $1 - F_i$ that integrates the recall R_i and false alarm rate F_i into one metric. The results show on the Juliet sub data set that the tools have not performed very well. The authors even stated that some tools are not even better than a random guess if a test case is vulnerable or not [40]. For comparison of the tools, they plotted Receiver Operator Characteristic (ROC) squares, where the x-axis was the false alarm rate and the y-axis was recall. Each plot represents a CWE category. They used the Friedman test to see if the tools had significant

differences in the performance. One significant difference between the two tools has been proven by the Friedman test.

As additional evaluation of the static code analysis tools, another evaluation has been done on three open-source projects. Three open-source projects with corresponding versions have been chosen that contain known vulnerabilities. Additionally, the version has been chosen where the vulnerabilities have been fixed. The same evaluation metrics from previous evaluations have been used. The results on the open-source project have shown that the static code analysis tool has performed a lot worse as on the Juliet sub data set.

Previous evaluations on static code analysis tools evaluate how the tools perform based on different data sets. In comparison, our contribution extracts specific source code patterns that are problematic for static code analysis tools. In addition, our research is based on vulnerabilities that occurred in real life projects. This ensures that those extracted patterns are actually code patterns that developers have written. Those patterns can be injected in source code to either create false negative or false positive reports from static code analysis tools. In addition, those patterns contribute that developers can mitigate them or static code analysis tools can be improved to detect them correctly.

4.3 Static code analysis models

There are many different static code analysis models. Many projects use machine-learning and deep learning algorithms to detect vulnerabilities [39, 32, 49, 79].

Nevertheless, Insecurity Refactoring requires a static code analysis model that is based on source code. Evans and Larochelle [37] describe their model to detect buffer overflows and format string vulnerabilities in C projects. Another attempt uses type qualifier to detect format string vulnerabilities in C projects [70]. Livshits and Lam [51] present their model that uses Java byte code and vulnerabilities described in Process Query Language PQL as input. A tainted object propagation uses a model to show how data can flow through the byte code. That model is used to track if data from a critical source will reach a critical sink. Their evaluation showed a low false alarm rate and they found 29 vulnerabilities in nine large open-source projects.

Graph-based models are popular because a program flow can be represented in a graph. Different approaches exist that use graphs to detect malware, e.g., [69, 33]. Hu et al. [42] present the Symantec Malware Index Tree (SMIT) that is used to efficiently determine the distance between graphs. They use it to determine the distance between graphs that represent malware

and function calls in binaries. That approach allows detecting malware based on the graph representation.

Those models are useful to detect security vulnerabilities. The goal of this thesis is to inject vulnerabilities. Accordingly, another model is required that allows to detect parts in the source code that can be insecurity refactored. The next section describes the Code Property Graph that is used as input for our model.

4.3.1 Code Property Graph

The approach of Insecurity Refactoring is based on the *Code Property Graph* [78]. The Code Property Graph is a combination of the *Abstract Syntax Tree* (AST), *Control Flow Graph* (CFG) and *Program Dependence Graph* (PDG). A *Property Graph* [64] is a graph structure that adds properties to nodes and edges. Yamaguchi et al. [78] define a *Property Graph*

$$G = (V, E, \lambda, \mu)$$

as a directed, edge-labeled, attributed multi graph. The set of nodes V, $E \subseteq (V \times V)$ is a set of directed edges, and $\lambda : E \to \Sigma$ defines the edge labeling function using the alphabet Σ to each edge. $\mu : (V \cup E) \times K \to S$ assigns the properties to edges and nodes where K is the set of property keys and S is the set of property values.

The *Abstract Syntax Tree* is defined as $G_A = (V_A, E_A, \lambda_A, \mu_A)$. It represents the Abstract Syntax Tree in the graph structure. Accordingly, a node $v_A \in V_A$ in the AST represents source code. For example, an addition in a program language can be a node. The operants of the addition are also AST nodes that are connected to the addition as edges $e_A \in E_A$. The corresponding code is added as a property value $s_A \in S_A$ assigned to the property key *code*. This allows to distinguish the different AST nodes. An AST is an ordered tree. This requires adding the ordering as another property to each AST node ($v_A \in V_A$).

The *Control Flow Graph* is defined as $G_C = (V_C, E_C, \lambda_C, \varnothing)$. A CFG is a graph that does not require any additional properties to represent the structure. The nodes V_C are the same statements and predicates as from the AST. The edge labels are defined as, $\lambda_C = \{True, False, \epsilon\}$ which represent if a condition has to be true or false. Accordingly, a edge in the CFG connects AST nodes together to represent how the program can flow through the code.

The *Program Dependence Graph* is defined as $G_P = (V_P, E_P, \lambda_P, \mu_P)$. It uses the same nodes as the AST and CFG. The PDG adds edges that represent control dependencies and data dependencies as known from the

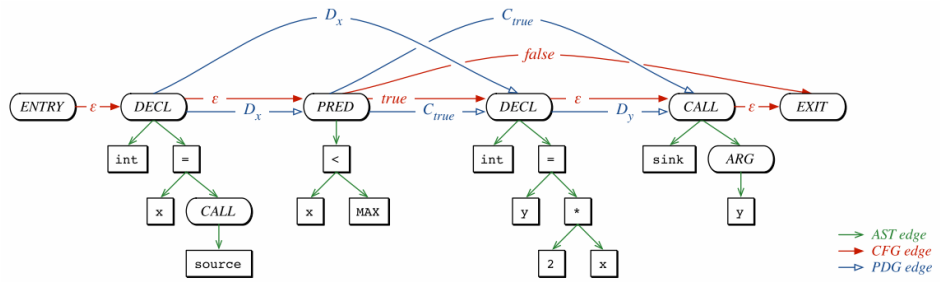


Figure 4.3: Code property graph from the source code sample. [78]

Program Dependence Graph. A control dependency defines what conditions have to be $True$ or $False$ to get to that statement. The data dependency defines where the variable value is assigned. This model allows tracking how the data flows and what dependencies are required.

A *Code Property Graph* combines these three graphs into one graph with the following definitions:

$$
\begin{aligned}
V &= V_A \\
E &= E_A \cup E_C \cup E_P \\
\lambda &= \lambda_A \cup \lambda_C \cup \lambda_P \\
\mu &= \mu_A \cup \mu_E
\end{aligned}
$$

Figure 3.2 shows the CPG for the source code sample from figure 3.1a. The AST Edges represent the Abstract Syntax Tree structure. The edges for the CFG and PDG are added to the AST structure to get a large graph that combines all representations. This allows to traverse the CPG in different directions. For example, the AST part can be used to detect specific function calls. Afterward, the function call can be used as a starting point for a control flow analysis.

Analyzing the *Code Property Graph* is based on traversing the graph. Yamaguchi et al. [78] define the *traversal* as a function $T : \mathbb{P}(V) \to \mathbb{P}(V)$ that maps a set of nodes to another set of nodes where $\mathbb{P}$ is the power set of V. The *traversal* is according to the *Code Property Graph* G. The symbol $\circ$ is used to chain traversals together. A filter traversal is defined as:

$$Filter_p(X) = \{v \in X : p(v)\}$$

This allows to filter a set of nodes X with the function p. The traversal from

one node to another node can be achieved with the following functions:

$$OUT_l(X) = \bigcup_{v \in X} \{u : (v,u) \in E \wedge \lambda((v,u)) = l\}$$
$$OUT_l^{k,s}(X) = \bigcup_{v \in X} \{u : (v,u) \in E \wedge \lambda((v,u)) = l \wedge \mu((v,u),k) = s\}$$
$$IN_l(X) = \bigcup_{u \in X} \{v : (v,u) \in E \wedge \lambda((v,u)) = l\}$$
$$IN_l^{k,s}(X) = \bigcup_{u \in X} \{v : (v,u) \in E \wedge \lambda((v,u)) = l \wedge \mu((v,u),k) = s\}$$

Additionally, traversals can be aggregated with OR and AND:

$$OR(T_1, ..., T_n)(X) = T_1(X) \cup ... \cup T_N(X)$$
$$AND(T_1, ..., T_n)(X) = T_1(X) \cap ... \cap T_N(X)$$

These are main traversal functions that are supported by common graph databases such as Neo4j [21] and InfiniteGraph [20]. Those definitions can be used to traverse the CPG to find vulnerabilities. Yamaguchi et al. [78] define a syntax-only vulnerability description $S = (M_0, M_1)$ where M_0 and M_1 are sets of $Match_p(X) = Filter_p \circ TNodes(X)$ traversals. A $Match$ function checks if a corresponding match is found on the Abstract Syntax Tree. This can be used to detect function calls, multiplications, etc. A syntax-only vulnerability is found if it matches traversals that are all in M_0 and it matches none that are in M_1. If the syntax is checked only, it will have a high false positive rate.

The Control Flow Graph can be included by the *control-flow vulnerability description* that is defined as a 4-tuple $(S_{src}, S_{end}, S_{dst}, \{(S_{cnd}^i, t_i)\}_{i=1...N})$ where:

S_{src} : set of Sources (syntax-only)
S_{end} : set of End-statements (syntax-only)
S_{dst} : set of Destinations (syntax-only)
S_{cnd} : list of syntax-only conditions and their outcomes

If a path from a source $v_{src} \in S_{src}$ to an end node $v_{end} \in S_{end}$ exists that does not reach a destination statement $v_{dst} \in S_{dst}$ in the path, a *control-flow vulnerability* is found. Additionally, for all nodes that match S_{cnd}, the corresponding label t_i must match. A control-flow vulnerability cannot be

Vulnerability types	**Code representations**			
	AST	AST+PDG	AST+CFG	AST+CFG+PDG
Memory Disclosure				✓
Buffer Overflow		(✓)		✓
Resource Leaks			✓	✓
Design Errors				
Null Pointer Dereference				✓
Missing Permission Checks		✓		✓
Race Conditions				
Integer Overflows				✓
Division by Zero		✓		✓
Use After Free			(✓)	(✓)
Integer Type Issues				✓
Insecure Arguments	✓	✓	✓	✓

Table 4.2: Vulnerability types and the possible code representations. [78]

detected if data is assigned to variables and the variable is then used in a critical function.

The last vulnerability description is the *taint-style vulnerability* description. It is defined as a 3-tuple $(S_{src}, S_{dst}, S^s_{san})$ where the set S_{src} defines attacker controlled sources, the set S_{dst} are sensitive sinks and the set S^s_{san} defines sanitization functions. All these representations are syntax-only representations (AST only). A *taint-style vulnerability* is found if there exists a path in the Program Dependence Graph from a source $s_{src} \in S_{src}$ to a sink $s_{dst} \in S_{dst}$ with the following two conditions. For each of Data Dependence Graphs also exist at least one path in the Control Flow Graph. The path does not pass a sanitization function $s_{san} \in S^s_{san}$. A taint-style vulnerability can be used to describe Buffer Overflows, Code Injections and Permission Checks, etc.

As evaluation, they decided to use the Code Property Graph on the Linux kernel source code. The evaluation has been done in two steps. In the first step, they have conducted a coverage analysis by reviewing vulnerabilities that have been reported on the Linux kernel in 2012. They have reviewed what types of vulnerabilities occurred and if they can be detected by the Code Property Graph. Table 4.2 shows the 12 vulnerability types that have been reviewed. Two of the reviewed vulnerabilities cannot be described in the Code Property Graph because *Design Errors* require additional information about the projects and *Race conditions* are difficult to be detected in a static analysis. A combination of all code representations allows describing the ten remaining vulnerability types. Other combinations can be used to detect some vulnerability types.

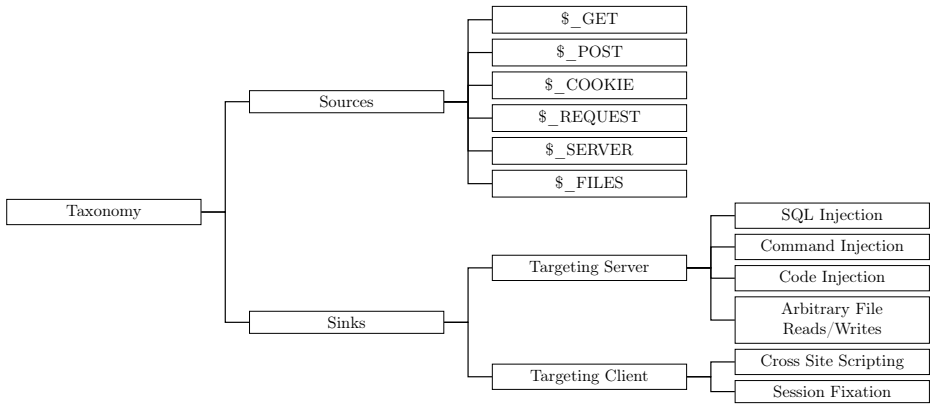


Figure 4.4: Taxonomy that is used to detect vulnerabilities in PHP. [27].

In the second step, they have written different rules for the different vulnerability types, including vulnerability types that did not occur in the Linux kernel in 2012. They have done a code analysis of the Linux kernel and have found 18 previously unknown vulnerabilities in the Linux kernel.

Backes et al. [27] used the Code Property Graph to detect vulnerabilities in PHP. They have implemented the concept of Code Property Graph for PHP by using PHP's internal parser to generate the AST. Then the AST has been used to generate the CFG and the PDG. They have extended the Code Property Graph with a Call Graph. This is a graph that contains edges from a function call to the corresponding function definition. All of these models are put together in a graph database to get a Code Property Graph. Figure 4.4 shows a taxonomy that has been used to detect the vulnerabilities in PHP projects. Sources are the basic function provided by PHP that obtains data from the user. The sinks are classified into either attacking the client or the server. For each vulnerability type, the related functions that fall into the category are listed. For each vulnerability, the different sanitization functions are listed. For example, for SQL Injection, the function *mysql_ real_ escape_ string* is one of the sufficient sanitization functions.

To increase the performance, the traversal of the Code Property Graph is split into two steps. The first step searches for all critical functions (sinks) in the graph and stores the corresponding IDs. The second step is the traversal of the Code Property Graph. A backwards taint analysis is done. Accordingly, the traversal starts at the sink and goes backwards using the PDG. If a sanitization function or a source is found, the traversal will stop. If a sink has multiple sources that reach the same sink, each source will create a different path. The paths represent the potential vulnerability and can be

reviewed to check if it is actually a vulnerability.

They evaluated the implementation on open-source projects from GitHub [19]. They searched for PHP projects that have at least 100 stars to ensure that the projects are relevant. They obtained 1854 projects in which four of these were projects that intentionally contained vulnerabilities. Accordingly, 1850 normal projects have been scanned. The reports of the normal projects have been reviewed to see if they are false positive or true positive reports and calculated the hit-rate $h = \frac{TP}{TP+FP}$. The results showed mixed hit rates. For *Arbitrary File Reads/Writes* and *Session Fixation* the hit rate was very low. For the other categories, the results show good hit rates (13.7% - 32%). Additionally, they compare the ratio of sinks to all function calls found in the projects compared to intentionally vulnerable projects. The ratio was always higher on the intentionally vulnerable projects in comparison to the normal projects.

Our contribution regarding static code analysis models is the Adversary Controlled Input Dataflow (ACID) tree. It uses the Code Property Graph to create the tree. The ACID tree provides a model that allows to detect vulnerabilities where the context of given input is relevant. For example, the context of a cross site scripting vulnerability has a high impact if that is exploitable. This makes the ACID tree precise in determining vulnerabilities and parts to inject vulnerabilities via Insecurity Refactoring.

4.4 Bug injection

Refactoring methods have been used to improve the security of source code. Thomas et al. [75] developed a tool that replaces database queries with prepared statements to remove potential SQL Injection vulnerabilities. Maruyama and Omori [52] present a security-aware refactoring tool. Research about injection of vulnerabilities in source code exists. Our contribution is to inject vulnerabilities using the Insecurity Refactoring definition. That definition requires that the normal usage of the program is still maintained. Other approaches try to minimize the impact on the normal usage by inserting triggers that only trigger on very specific inputs. In contrast, the vulnerabilities injected with Insecurity Refactoring will create a vulnerability that developers unintentionally create. For example, a sanitization function is used that is insufficient for the given sink. This allows to exploit the vulnerability by using critical characters that are not filtered out by the sanitization function. LAVA and EvilCoder that are described in the following sections. Those tools are injecting vulnerabilities in Java and C code. Our injected vulnerabilities are using source code patterns that have been extracted from

existing vulnerabilities. The injected vulnerabilities from Insecurity Refactoring do not require an artificial specific input instead it requires bypassing the sanitization functions as it happens in real vulnerabilities.

4.4.1 Large-scale Automated Vulnerability Addition LAVA

Dolan-Gavitt et al. [35] developed the Large-scale Automated Vulnerability Addition (LAVA) tool. The tool uses a dynamic taint analysis-based technique to automatically inject vulnerabilities in C projects. For the injection, they use DUAs (Dead, Uncomplicated and Available Data). A DUA is a user controlled input that does not change any control flows and is not concatenated with other variables. A DUA is determined by the *taint compute number* (TNC) and the *liveness*. The TNC is computed by all concatenations that define the variable. For each concatenation, the TNC is iterated once. Additionally, the TNC is iterated up for each run through a loop. The *liveness* is calculated by counting in how many control flow decisions the variable is used. Accordingly, the perfect DUA is not used in any concatenations (uncomplicated) and will not be used in any control flow decisions (dead).

The next important part is the *Attack Point* (ATP). This is a code location that can be transformed into a vulnerability. The focus relies on Buffer Overflow and read/write out of bounds vulnerabilities. Corresponding functions like *memcpy()* are an *Attack Point*.

The injection of the vulnerability is approached by searching for DUAs that are near *Attack Points*. This allows to transform the *Attack Point* by using the DUA variable. In this approach, it is not required that the variable is used in the *Attack Point*. If the DUA is not in the context of the ATP, new code is added that makes the DUA available (static or global variable). In their concept of LAVA, a vulnerability should only be exploitable in a narrow case and the vulnerability should not constantly occur. This makes the injected vulnerabilities behave like non injected vulnerabilities. Their approach to the narrow case is by adding checks that only occur on specific bytes.

Figure 4.5 shows a code example that contains a DUA (variable b) and an *Attack Point* (*memcpy*). The variable b is a DUA because it is not a result from a concatenation and is not used in the *if* statement. The program language comment shows how the tool would inject a vulnerability. It uses the variable b and checks for a specific byte. If b equals to that byte value, it will add itself to the variable d that will create a buffer overflow vulnerability.

The implementation of the framework uses four steps. Figure 4.6 shows the implementation architecture of LAVA. The first step is instrumenting

```
void foo(int a, int b, char *s, char *d, int n) {
    int c = a+b;
    if (a != 0xdeadbeef)
        return;
    for (int i=0; i<n; i++)
        c+=s[i];
    memcpy(d,s,n+c); // Original source
    // BUG: memcpy(d+(b==0x6c617661)*b,s,n+c);
}
```

Figure 4.5: Code example of a LAVA injected vulnerability. [35]

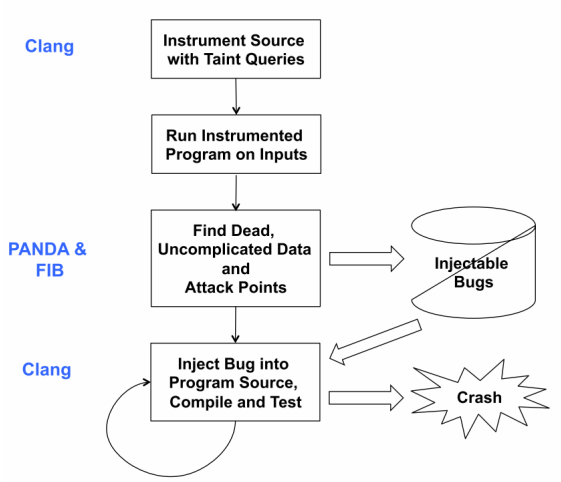


Figure 4.6: Lava implementation architecture. [35]

the source code with taint queries. This allows to analyze at the dynamic testing to check where the input will be stored and read. In the next step, the program will be run with different inputs. Each byte in the input has a unique ID. This allows to identify what specific bytes are stored and read. In the third step, the DUA and ATP are searched. A DUA is found based on threshold values for the TNC and *liveness*. Different vulnerabilities can be injected based on the DUAs and *Attack Points*. In the last step, the vulnerabilities are injected and the modified program is compiled to ensure that the syntax is still correct.

They have done three different evaluations for their approach. The first evaluation checks if open-source projects can be used to inject vulnerabilities. Additionally, it shows how many vulnerabilities can be injected and a test exploit shows if the vulnerabilities are exploitable. Four open-source projects

have been used to inject vulnerabilities. The results indicate that many vulnerabilities can be injected. They count each vulnerability as a different vulnerability if the locations of the DUA and *Attack Point* are different because the path in between can be different.

As second evaluation, they have checked the distribution and realism of the vulnerabilities. A good distribution on the project means that detection tools have to analyze all these parts to find all vulnerabilities. The bug realism is calculated by using the code positions of the DUA and ATP, the histogram of the positions and the trace between the DUA and ATP. For example, if static or global variables have to be added to pass data to the ATP, the sample would not be realistic. If the DUA and ATP are near together, a sample has been defined as realistic.

As the last evaluation, a fuzzer and symbolic execution-based bug finder (SES) have been used to see if they can detect the injected vulnerabilities. They have used two different triggers for the vulnerabilities. The *Knob-and-trigger* requires two different bytes to be a specific value to trigger the vulnerability. The second trigger is *Range*, where a byte has to be in a specific range to trigger the vulnerability. For *Knob-and-trigger*, only the SES could find them. For *Range*, the SES performed very similar on range because the specific byte is important to statically detect a vulnerability. As expected, the fuzzer performs better on larger trigger ranges. A second test data set with the name *LAVA-M* has been created. In this data set, as many vulnerabilities as possible have been injected in four open-source projects. The tools have been used again to scan the data set and both tools did not perform very well. At least, the results from the two tools have only overlapped a little. Accordingly, the LAVA data set is not tailored to a specific finding strategy.

Hulin et al. [45] used the LAVA application to create challenges for a capture the flag (CTF) event. Firstly, they had to improve the LAVA application to actually be able to inject exploitable bugs. Two exploitable vulnerability types, *direct stack pointer corruption* and *controlled relative memory writes* have been implemented for the LAVA application. Additionally, a domain-specific language has been implemented that describes the injection patterns. The pass from a DUA to an ATP has been modified as well. Instead of unique global variables, an array called *Dataflow* will be used. Each element stores that array to be used for a DUA, ATP pair. The magic value to trigger the vulnerability is still implemented for the injected bugs to maintain the functionality on benign inputs.

For the CTF event, eight challenges have been prepared. Two applications have been developed in C that have simple functionalities. These applications

have been used to inject exploitable bugs with LAVA. Each challenge is a copy of the initial application modified with the injected bug. Additionally, non-exploitable bugs (chaff bugs) have been added by the LAVA tool because the combination of the magic value and the *Dataflow* variable makes it obvious where injected bugs are. The combination of exploitable and non-exploitable bugs makes it a challenge to determine what bug is exploitable. Four challenges have been created by using the LAVA tool. Additionally, four challenges have been manually created.

The CTF event has been held for one week and four university security clubs have attended. The clubs have been interviewed to evaluate the usage of LAVA injected bugs as CTF challenges. For many participants, the challenges have been too difficult. Especially for the students who are relatively new to reverse-engineering. Additionally, the repetition of the same projects has been stated positive and negative. The positive aspect is that it made reverse-engineering easier because the attendees got familiar with the code. In contrast, it makes the challenges repetitive and that gets boring. Some attendees would have preferred only one project that contained all the vulnerabilities. The reverse-engineering experienced students had no problems with the reverse-engineering part, but had problems at developing the exploits. Accordingly, they have seen the challenges as a good opportunity to train exploit development. The magic value has been stated as negative. It makes the challenges abstract and makes the finding of the vulnerabilities easy. Some attendees were even able to identify patterns of the non-exploitable bugs which allowed them to distinguish between exploitable and non-exploitable bugs.

The magic value has been stated as a limitation that should be mitigated for better bug injection. Additionally, the exploitable bugs have been tested by the hosts of the CTF to ensure that they are actually exploitable. All of them were exploitable, but the testing took a lot of time compared to the automatic bug injection.

Compared to our contribution, LAVA injects vulnerabilities that trigger on specific inputs. In addition, it changes the normal usage of the program by using DUA. It minimizes the impact on normal usage but does not maintain it like our approach. In the code example of figure 4.5, the LAVA injected vulnerability will trigger if the user input is *0xdeadbeef*. It will trigger a buffer overflow by modifying the destination of *memcpy*. The authors make no insurance that the variable value of *0xdeadbeef* is not a valid input. Accordingly, it may change the normal behavior of the program but they reduce the likelihood by using DUAs. This makes the vulnerability like a backdoor that only users can exploit that have knowledge about it. In

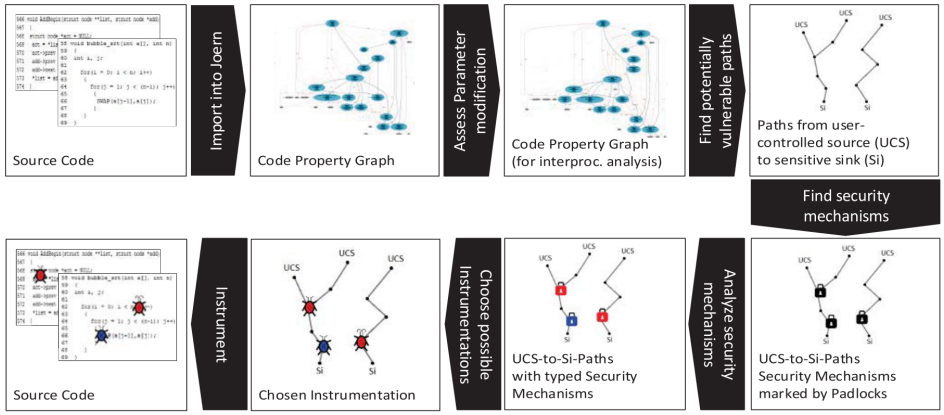


Figure 4.7: Workflow of the *EvilCoder* application [61].

addition, the DUA is connected to an attack point. This connection is not supposed to be in the code and can result in unintentional bugs that also may change the external behavior. Furthermore, the injected trigger points are easily detectable if someone reviews the source code. In contrast, our approach uses source code patterns from real software projects and those are not that prominent in a code review.

4.4.2 Bug injection by using the Code Property Graph

Pewny and Holz [61] developed the *EvilCoder* application. The main concept is to find potentially vulnerable source code locations and modify them that a vulnerability exists. Figure 4.7 shows the workflow of the project. The initial source code is used to create the Code Property Graph. The creation of the CPG is based on the island grammars Moonen [55] to parse the C source code. The CPG does not support interprocedural analysis. Similar to Backes et al. [27], a call graph is added to the CPG. Additionally, the parameters are checked if they return data by using references or pointers. This creates the Code Property Graph that is ready for interprocedural analysis. The *glibc* library has been added to the Code Property Graph to resolve functions from that library.

The potential vulnerable locations are found by a backwards data flow analysis. It starts by searching for potential sinks. For each vulnerability type, different sinks are searched. For example, the *memcpy* function is a Buffer Overflow sink. The traversal starting from the sink is a tree-like structure. The interprocedural analysis requires distinguishing between the following five cases:

1. **Increment/Decrement**
   E.g., *i++;*
   An increment or decrement of a variable requires continuing analyzing the same variable.

2. **Left-hand-side of arithmetic expression**
   E.g., *a = b + (c * 2);*
   All the variables that are found on the right sight will be further analyzed. In the example, it would be *b* and *c*.

3. **Assigned as return-value of function-call**
   E.g., *c = f(a, b)* and *int f(int x, int y){int r = x+y; return r;}*
   In this case, the data flow analysis continues at the return statement with the variable *r*.

4. **Assigned as argument in function call**
   E.g., *strcpy(dst, src);*
   Functions that are resolved by the Code Property Graph will continue at that parameter that will then end as another parameter. For external functions, a "data-transfer" lookup-table is created.

5. **Assigned as parameter of a function**
   E.g., *f(int x){a=x;}*
   If the analysis is at the variable *a*, the next variable to track would be variable *x* where it stems from the parameter of a function. Then all function calls are searched up and the analysis continues at the corresponding parameters.

All paths from a user controlled source (UCS) to a sensitive sink (Si) are potentially vulnerable locations. They define user controlled source that is project-specific if the data is from files, network, command line, standard input streams or environment variables.

To find the security mechanism, the Control Flow Graph is searched for security checks. Between each data flow node, the Control Flow Graph is searched for control flow nodes. If a control flow node is a security check, it will be determined by heuristics based on *return* and *exit* statements.

The *instrumentation* injects the bug by changing the security checks. First of all, the instrumentation checks if the security mechanism is understood correctly. If it is not understood correctly, the tool cannot transform the security mechanism. If it is understood correctly, all applicable instrumentations are listed and one is chosen randomly to inject the vulnerability.

The security checks can be replaced with the following instrumentations:

	libpng	**vsftpd**	**wget**	**busybox**
Lines of code	40,004	20,046	137,234	265,887
User-controlled sources (UCS)	9	3	21	152
Sensitive sinks (Si)	98	13	453	573
Unique UCS-Si combinations	158	22	22	30
UCS-to-Si data-flow paths	22,516	786	1,882	2,905

Table 4.3: Results from automatic bug insertion [61].

- Remove the security check
- Surround the security check with another check that always fails
- Arithmetically influence the decision logic
- Move the security check into an unrelated path
- Swap the security check and the sink
- Use security antipatterns for integer overflow checks

For example, a length check ($length > 512$) can be replaced by arithmetically influencing the decision logic ($length/2 > 512$).

The evaluation of the approach has been done on four open-source projects *libpng*, *wget*, *busybox* and *vsftpd*. Table 4.3 shows the result from automatic bug insertion on the open-source project. All the projects can be used to inject bugs. As the unique UCS-Si combinations have different data flow paths, numerous bugs can be injected. The injected bugs cannot ensure that they are exploitable. The large amount of control flow checks might miss additional security checks that prevent the exploit.

Pewny and Holz [61] have a similar approach in the iteration of the Code Property Graph that is used in the Insecurity Refactoring approach. It also does a backwards data flow analysis that results in a tree-like structure. That is the nature of a backwards data flow analysis. In addition, function calls require a straightforward approach. Accordingly, the backwards data flow analysis approach is similar to our approach. The Insecurity Refactoring approach creates the ACID tree. It includes the concatenations and excludes nodes to allow a precise context analysis. As the approach from Pewny and Holz [61] only considers security checks based on length checks, such a context analysis is not required. The refactoring from Insecurity Refactoring injects vulnerabilities that are exploitable based on the context, sanitization function and sink [50] [65].

4.5 Static code analysis test suites

Test suites for static code analysis tools are usually synthetic or based on known vulnerabilities. The Juliet [29] test suite is a synthetic test suite that contains 28,881 Java test cases and 64,099 C/C++ test cases. The test cases cover many different CWE categories. The STONESOUP test suite [60] is a combination of synthetic and open-source projects with seeded vulnerabilities that are split into three phases. The first two phases contain small test cases (synthetic) and the third phase contains vulnerabilities that are seeded in source code.

4.5.1 PHP Test Suite

Stivalet and Fong [74] developed a tool to create a large test suite for static application security testing (SAST). The tool is reusable and easy to use without any prior knowledge. The design allows adding custom rules to generate additional test cases. The initial design of the test cases is based on the OWASP top ten 2013 [10]. Test cases with flaws are used to test the tools for false negative reports. In contrast, test cases with correct code are used to test for false positive reports.

Figure 4.8 shows the internal structure of the tool. Each test case uses a selected *input*, *filtering* and *sink* template. The *input* template provides untrusted data that can be manipulated by an attacker. The *filtering* template does some filtering on the untrusted data. The filtering can be sufficient or not that either result in *safe* or *buggy* code. The *sink* template contains sensitive operations where untrusted data can trigger a vulnerability. To introduce complexities, different complexities can be chosen for each test case. A complexity can be either a condition, loops, functions, classes or multiple files. Each template type can get a complexity. If a complexity is chosen, the templates will be put inside these complexities. For example, a loop is chosen for the sink, the sink template will be put inside a loop. The templates are stored in XML files that contain different attributes and the corresponding source code for each template.

Each test case is composed of the selected templates. Figure 4.9 shows a sample from the PHP test suite. No complexity has been used in that sample. User data is provided with the *$_POST* functions from the *input* template. It is filtered with *mysql_real_escape_string* from the *filtering* template. As *sink* template, a SQL query has been used. For each test case, a *manifest.xml* file is generated that contains information about the test case. For example, if a test case is safe or unsafe and a list of files from the test case.

Table 4.4 shows a summary of the generated test cases. It covers six

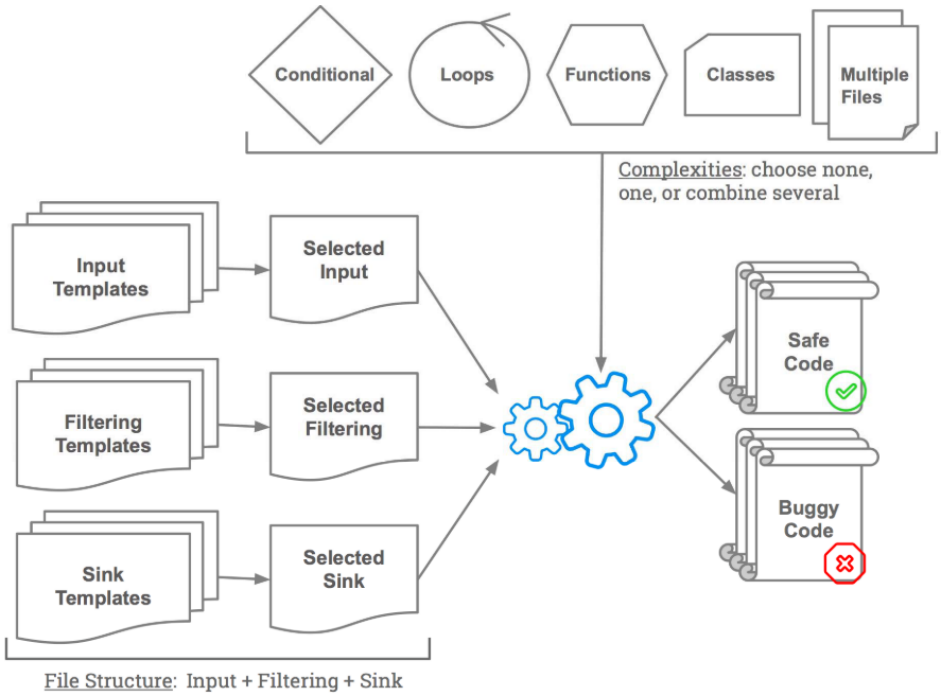


Figure 4.8: The internal structure [74].

```php
<?php

    $input = $_POST['UserData'];                                    Input

    $tainted = mysql_real_escape_string($input);                    Filtering

    $query = "SELECT * FROM student where id=". $tainted . "";

    $conn = mysql_connect('localhost', 'mysql_user', 'mysql_password');
    mysql_select_db('dbname') ;
    echo "query : ". $query ."<br /><br />" ;

    $res = mysql_query($query); //flaw                              Sink

    while($data = mysql_fetch_array($res)){
        print_r($data) ;
        echo "<br />" ;
    }

    mysql_close($conn);
?>
```

Figure 4.9: Sample generated vulnerable PHP test case [74].

vulnerability categories containing 29,258 safe and 12,954 unsafe test cases. Cross Site Scripting and SQL Injection have the most test cases. In total, 42,212 test cases have been generated based on the developed templates. Our contribution is more precise in determining if a test case is vulnerable by using decision trees. Their test suite has been reviewed and missing parts have been added to our data set. Especially, the patterns that are problematic for static code analysis tools are not found in their data set. Accordingly, our generated test suites are more precise and an extension to their test suite.

4.5.2 An explainable benchmark

Hao [11] provide a method to construct explainable benchmarks based on vulnerabilities in projects. The goal is to have a representative benchmark (reference to real-world settings), measurable (quantitatively measurable) and explainable (explain the capabilities of the tested tool). The idea is to use the initial source code containing the vulnerability. Then the vulnerability related code will be extracted. Then the source, the sink and syntactic features that are involved in the vulnerability will be identified. Based on those patterns, different test cases can be generated. Figure 4.10 shows the construction of the benchmark on open-source projects. The first step is to extract the vulnerability related code. The extraction is function-based to maintain the functionality of the extracted source code. At first, logging instructions are added at the beginning of all functions found in the source code. Then a proof of concept (PoC) is run to trigger the vulnerability and the corresponding logging statements are stored. After the PoC, all statements will be removed where corresponding function bodies have not been called. Accordingly, only the functionality of the PoC will be maintained. The reduced source code is the *original* test case.

An additional test case is the *basic* test case. Such a test case contains only the corresponding source and sink of the *original* test case. The *basic* test case is constructed by searching for the source and sink. The sink can usually be identified by looking into the corresponding CVE report. The source is found by using the *rr* debugger tool [18] to perform a backwards taint analysis. The basic test case is then constructed by combining the source and sink.

An explainable benchmark test suite requires finding out the reason for false negative reports. The idea of the authors relies on *features*. Table 4.5 shows the features they have identified based on previous work ([63] [54]). The features influence the control flow and the data flow part. The authors assume that the static code analysis tools search for a sinks, source and then

Vulnerability	**CWE**	**Safe**	**Unsafe**	**Total**
Insecure Direct Object Reference	862 - Missing Authorization	400	80	**480**
Injection	72 - OS Command Injection	1,872	624	**2,496**
	89 - SQL Injection	8,640	912	**9,552**
	90 - LDAP Injection	1,728	2,112	**3840**
	91 - XML Injection	4,784	1,264	**6,048**
	95 - File Injection	1,296	336	**1,632**
	98 - PHP Remote File Inclusion	2,592	672	**3,264**
Sensitive Data Exposure	311 - Missing Encryption of Sensitive Data	2	2	**4**
	327 - Use of a Risky Cryptographic Algorithm	3	5	**8**
Security Misconfiguration	209 - Information Exposure Through an Error Message	5	3	**8**
URL Redirects and Forwards	601 - URL Redirection to Untrusted Site	2,208	2,592	**4,800**
Cross Site Scripting	79 - Cross-Site Scripting (XSS)	5,728	4,352	**10,080**
	Total	**29,258**	**12,954**	**42,212**

Table 4.4: Summary of the generated test cases [74].

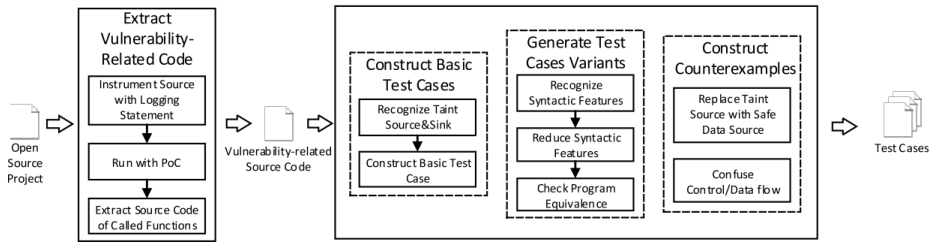


Figure 4.10: Benchmark construction overview [11].

check if any connections between them exist. These connections are based on the features. The test case *variants* are created based on these features. A data flow analysis on the *original* test case is used to search for syntactic features. For each syntactic feature that is found, a test case is generated that does not contain the feature. For example, if a *function pointer* feature is found, it will be replaced with a direct function call. Each test case *variant* is checked for program equivalence. A proof of concept is run on the variant and the resulting memory state is compared with the memory state from the *original* test case by using the Clang's AddressSanitizer [12]. If the same error type and same stack trace is found, the test case variant will be added to the benchmark.

The final test cases are *counterexamples*. These test cases do not contain vulnerabilities. For each test case *variant*, a *counterexample* is created by adding a secure source. If a control flow syntactic feature is found in the *variant*, the control flow from the initial sample to the sink is blocked and a flow from the secure source to the sink is added. If a data flow syntactic feature is found, the secure source and initial source are mixed in the data flow. The mixing ensures that the data from the initial source cannot reach the sink. For the *basic* test case, a $counterexample_{base}$ is created by replacing the source with a secure source.

These different test cases allow identifying what source, sinks and syntactic features the tested tools support. The *basic* test case shows that the tool supports the vulnerability. The $counterexample_{base}$ shows if the tool checks for insecure data. The *variant* test cases can be used to check what syntactic features the tool supports. This makes the benchmarks explainable and measurable. The test cases are based on existing vulnerabilities which make the benchmark representative.

The approach has similarity to our approach of creating a static code analysis test suite. Those are also based on existing vulnerabilities and they also used an extraction method to extract vulnerable source code.

Syntactic Features	**Influence on SCA**	
	Control Flow	***Data Flow***
Function Pointer (FP)	✓	
Function Call Chain (FCC)	✓	✓
API Function Call (AFC)		✓
Data Structure (DS)		✓
Data Array (DA)		✓
Data Pointer (DP)		✓

Table 4.5: Syntactic/Semantic Features in C language [11].

Our contribution is that all the specific parts are described in a pattern language. In combination with the defined vulnerability decision tree, almost all permutation of those patterns can be used to create different test cases. The test case generation has been evaluated by experts from SAMATE [5] to ensure solid test cases.

Chapter 5

Methodology

This section describes the methodology. The scientific questions are split into five categories (Q1-Q5). To answer the corresponding questions, different choices of methods were used. The following methodologies are mapped to the corresponding scientific questions. For example, the choice of method M2 is to answer the scientific question Q2. Figure 5.1 shows the relations of the methodologies. The Insecurity Refactoring requires the pattern of vulnerabilities and the patterns of source code patterns that are problematic for SCA tools. Based on M1 and M2, the Insecurity Refactoring can be implemented. In methodology M4, three different methodologies have been used to evaluate the Insecurity Refactoring approach. In methodology M5, two different static code analysis test suites have been created. An expert interview has been used to evaluate the generation process of the test suites.

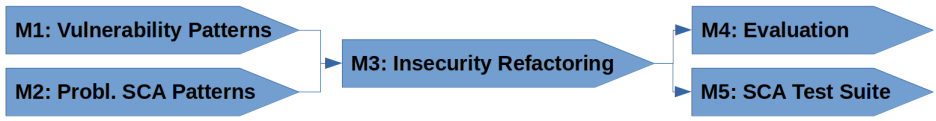


Figure 5.1: Overview of the methodologies.

5.1 M1: Classification of source code patterns in open-source projects

The classification of source code patterns is a qualitative research method that identifies source code patterns and categorizes them based on manual code analysis of existing vulnerabilities. Compared to prior work, this methodology aims to classify the source code so that those patterns can be transferred back into source code again. Previous work tried to classify the vulnerabilities instead of the source code that resulted in a vulnerability. In addition, the manual review also aims to identify pitfalls that software developers can make that result in a vulnerability. Figure 5.2 shows the

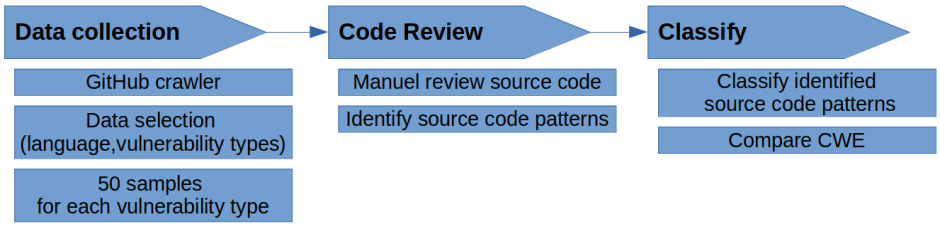


Figure 5.2: Methodology to classify source code patterns.

steps to classify the source code patterns of existing vulnerabilities. The data collection should be based on vulnerabilities that occurred in real projects. Accordingly, the database CVE [1] has been chosen because it contains almost all vulnerabilities that occurred in software projects. In addition, the database contains the information on how such security issues occurred in real projects. The analysis of source code patterns requires access to the source code. Accordingly, the first step is to use a crawler to find all CVE reports that are categorized into vulnerability types and have a relation to open-source projects. Based on these findings, important vulnerability types and programming languages are chosen as a data set. The data set is obtained by a crawler tool that searches for free accessible source code, that is related to CVE reports. The source code of the data set is manually analyzed to identify different source code patterns. A manual analysis of the given vulnerability is required because no modern static code analysis tool can ensure a perfect detection rate. As a full source code review would be almost impossible, the CVE report data is used to decrease the review process only on vulnerability related source code. An automatic approach will require to implement a similar procedure as static code analysis does. Accordingly, an automatic approach would provide the same problems as

static code analysis tools have to fight with. Especially source code patterns that are difficult for static code analysis tools are interesting and will not be found by an automatic approach.

After the source code from the vulnerabilities have been collected, the code review phase identifies source code patterns. At first, the manual analysis classifies the source code patterns found in the data set. The focus relies on vulnerabilities that depend on malicious data reaching critical functions. This allows to review the source code from a given source to a given sink. Based on the data set, we decided to analyze SQL Injections, Cross Site Scripting and Buffer Overflow vulnerabilities manually. For each sample, the common known parts (source, sanitization, sink,...) of such vulnerabilities are classified. The information about the source is provided by the CVE report. Based on the vulnerability type, the corresponding sink can be identified. Additionally, it is analyzed if on the path any important data flow paths were involved. Important data flows are paths where data flows from one part to another part. For example, an environment variable could be set in one part and then that environment variable will be read in another part. Such patterns are also tracked in the review phase. Suitable classifications based on the analysis results are constructed. Again, in comparison to previous work, the perspective is the software developer and the categories must be able to transform back into source code. Besides the classification of the vulnerability, the research also includes how the developers resolved the issues. This requires analyzing the patch provided by the developers to fix the vulnerability. A comparison to the Common Weakness Enumeration (CWE)[9] shows if any classifications are missing as detailed as our results.

5.2 M2: Limitations and problems of static code analysis tools

A mix of quantitative and qualitative study is used to find the limitations and problems of static code analysis tools. Figure 5.3 shows the methodology to extract the problematic source code patterns. In comparison to previous static code analysis evaluations, the goal is to find source code patterns that static code analysis tools cannot analyze correctly. Our previous work already provided a database of vulnerable source code. The methodology is to scan the previous database and the corresponding patched version to identify which source code patterns static code analysis tools cannot detect. At first, it requires selecting static code analysis tools that are used for identifying the problematic source code patterns. For this methodology, commercial and open-source static code analysis tools are chosen because they match the

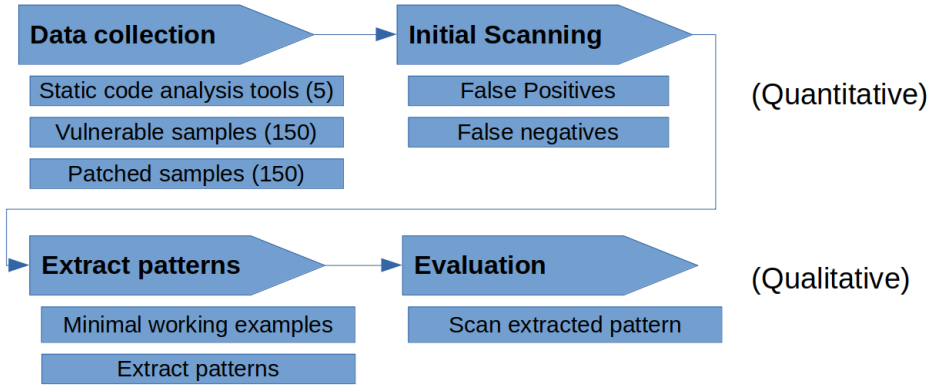


Figure 5.3: Methodology to extract problematic source code patterns.

criteria to (1) support PHP projects, (2) searches for security vulnerabilities, (3) support data flow analysis and (4) are still maintained (last update $<$ 1 year). Two open-source and three commercial tools are selected for this study that match the criteria. The goal of this methodology is not to compare the tools and the license for the commercial tools did not allow publishing the names. Accordingly, the commercial static code analysis tools are called Tool A, Tool B and Tool C. The goal is to find source code patterns that are problematic for state-of-the-art static code analysis tools where anonymized names are not a problem. Accordingly, five static code analysis tools are selected.

A process consisting of four steps is used to find source code patterns that are problematic for static code analysis tools. The steps are (1) selection, (2) scanning, (3) identifying patterns, and (4) verification. The (1) selection of a data set uses the same vulnerabilities from the previous data set. This makes the review phase simpler because the vulnerabilities have already been reviewed and the vulnerability location is known. Nevertheless, it only contains the vulnerable source code. False positive reports from static code analysis tools are a problem as well. Accordingly, we decided to also include the patched source code of the vulnerabilities. The data set is split into a vulnerable data set that contains source code including the vulnerability and a fixed data set that contains the source code that is patched to fix the vulnerability. Then the data sets are (2) scanned by the static code analysis tools. The location of the vulnerabilities are known by the previous manual code analysis. This allows to automatically identify false negative and false positive results. The vulnerabilities and patched versions that provided such

false reports are interesting to find out why they provide such false reports. Accordingly, the next step (3) is required to identify what is the problematic source code pattern. Manually reviewing the full project would not be efficient and would take too much time. At first, the identifying pattern step uses all false negative and false positive results and extracts a minimal working example of the vulnerability. The minimal working example contains only the source code that corresponds to the vulnerability. These minimal working examples are scanned again to validate that the problematic patterns are included. If the minimal working example did provide a false report from a tool, the problematic pattern has been missing and the minimal working examples has to be increased until it contains the problematic pattern again. Then the minimal working examples are reviewed manually to find all source code patterns that are used. The (4) verification step is required to verify which of the found source code patterns are actually problematic because the extracted patterns are just guesses based on the manual review. To verify the patterns, a small source code sample is used that contains a basic vulnerability and is detected correctly by the five static code analysis tools. For each identified source code pattern, the sample is modified to contain the problematic pattern. Accordingly, if a tool cannot identify the vulnerability anymore, a problematic source code pattern has been identified that produces false negative reports. In contrast, patterns that are problematic for false positive reports the modified source code does not contain the vulnerability anymore. For example, a sanitize function is used that filters all characters except of numbers. Consequently, the last steps verify what source code patterns are problematic for static code analysis tools.

5.3 M3: Insecurity Refactoring

The goal of Insecurity Refactoring is to inject vulnerabilities in projects. The process for the implementation is shown in figure 5.4. Based on its

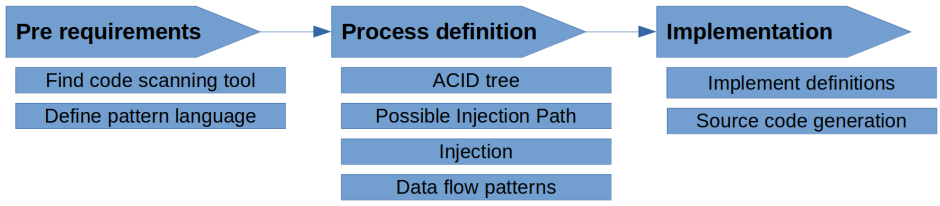


Figure 5.4: Implementation overview.

definition, a process has been created and defined that uses static code analysis approaches to identify possible injection paths that can be transformed into a vulnerability. The programming language considered is PHP because most of the identified patterns stem from PHP. Reviewing open-source static code analysis projects showed that the Code Property Graph (Yamaguchi et al. [78] [27]) supports PHP and can perform a data flow analysis that is required for our approach. Accordingly, this approach uses the same static code analysis model to detect possible injection paths as EvilCoder [61]. In comparison to their work, our approach uses source code patterns that have been identified and classified from our previous work. For the detection part, the previously identified source code patterns have been extended with patterns from the standard PHP documentation [62]. For example, all different SQL driver libraries have been added as possible SQL Injection sinks.

The Insecurity Refactoring approach is based on our defined Adversary Controlled Input Dataflow (ACID) tree. Creation rules define how an ACID tree is created by using the Code Property Graph and a given sink as inputs. The ACID tree is a tree structure where the sink is the root node and all the leafs show possible inputs that can reach the sink. The ACID tree can be used to find vulnerabilities or to find possible injection paths (PIPs) by traversing from a leaf to the sink. A possible injection path means that a leaf provides user input data and the sink is a critical function. Instead of a vulnerability, there must be at least a sanitization function, a secure sink or secure source involved. A secure sink means that it is not possible to perform an attack on that sink. For example, a prepared statement is correctly used by setting the parameter of a SQL prepared statement. Similarly, a secure source means that the source only provides uncritical characters (e.g., only numbers) that cannot be used to exploit a vulnerability. PIPs are used for the Insecurity Refactoring process to transform a non-vulnerable source code part into a vulnerable source code part. Transformation rules are defined to inject a vulnerability without breaking the normal usage of the program. Accordingly, the sanitization functions, secure sink and secure source are replaced to be insecure. Insecure means that the function does not provide sufficient protection against exploits. The definition if a sanitization function is sufficient is based on the sanitization function, the context where the input is located and the sink. For example, the context means that the user input is put inside apostrophes. Based on the sanitization function and sink, it might be sufficient to filter out all apostrophes. Accordingly, based on the context and sink a insufficient sanitization function can be chosen. The difference to the injection from LAVA [35] and EvilCoder is that those tools injected vulnerabilities that can be triggered given a specific input. For example,

an input requires being *0xdeadbeef* to trigger the injected vulnerability. In contrast, our approach does not require such specific inputs. Instead, it is based on source code patterns that have been identified from the reviewed vulnerabilities.

To represent all the previously defined source code patterns, a PL/V pattern language has been defined that represents the source code patterns. This is a context-free language, that allows to describe previously defined source code patterns in a program independent language. Based on the language, all the source code patterns can be defined and it allows checking the ACID tree if such a pattern exists in the tree structure. The language would even allow representing such input triggered vulnerabilities like LAVA and EvilCoder implemented. The idea of Insecurity Refactoring is to create learning examples. By just replacing sanitization methods with insufficient methods, the variance of learning examples would be only mainly based on the code base that will be used. Accordingly, we added the support to insert data flow patterns. Those patterns allow inserting different data flows, but each start and end point of the data would be the same. In other words, a data flow pattern will not change the data flow instead it will just flow different than before. Those data flow patterns are used to represent the difficult source code patterns for static code analysis tools. LAVA also changes data flow, but they use it to connect data flows that did not exist before. Accordingly, if such a changed data flow is required, the normal usage of the program may change. It might result in a crash of the program, even if the vulnerability is not exploited.

All the modifications of Insecurity Refactoring are done on the ACID tree. Such a modified ACID tree will not automatically result in source code. In the last step, we implemented a transformation of the modified ACID tree to the corresponding source code. All the nodes in the ACID tree store the corresponding code location. For each modified part of the tree, the related source code has to be replaced with the modification. The ACID tree is based on the Code Property Graph which also includes the Abstract Syntax Tree. The code modifications are translated into source code based on the underlying Abstract Syntax Tree in combination with the PL/V pattern that also can be translated into an Abstract Syntax Tree. Then the translated source code will simply be replaced in the corresponding lines of code of the source code.

5.4 M4: Evaluation of Insecurity Refactoring

Figure 5.5 shows that the evaluation is split into two parts. The first part evaluates if the Insecurity Refactoring definition holds on real projects. The second part evaluates if Insecurity Refactoring can be used as learning examples.

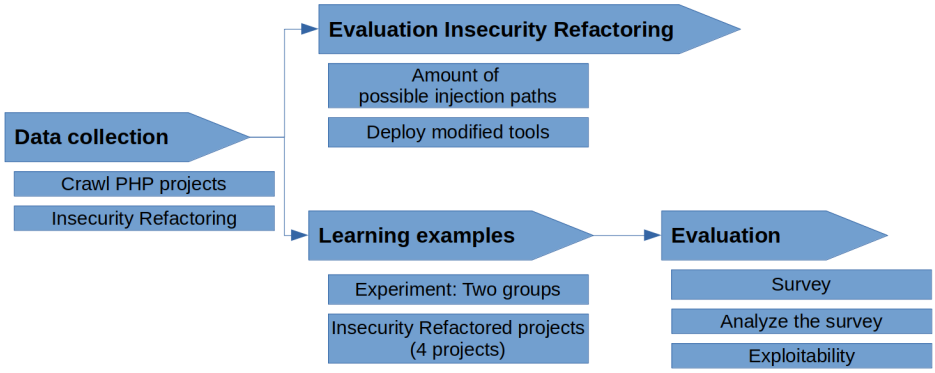


Figure 5.5: Evaluation of Insecurity Refactoring.

To see if our approach of Insecurity Refactoring works on actual projects, a quantitative research on open-source projects is used. It will be checked on how many open-source projects the Insecurity Refactoring process is possible. An Insecurity Refactoring process is possible if the implemented tool finds a possible injection path. Accordingly, open-source projects from GitHub are crawled and scanned for possible injection paths. The results show for how many projects it is possible to inject vulnerabilities by the previous defined Insecurity Refactoring process. Just finding a possible injection path does not show if the process breaks the normal usage of the program. The evaluation should show if the Insecurity Refactoring did not break the normal usage. This requires evaluating projects that have been modified by the Insecurity Refactoring process. We decided to evaluate the normal usage via two experiments that also will be used to evaluate the usage as learning examples. If there are any complaints about the program not running as usual, we know that the Insecurity Refactoring did break the normal usage of the program.

Two experiments on different groups are used to evaluate the usage of the program and if the insecurity refactored vulnerabilities can be used as learning examples. For the experiment, four different open-source projects are modified by the Insecurity Refactoring process. These projects are deployed

to virtual machines. The small number of open-source projects is a result of the setup of such a project taking a lot of time to get them running in the first place. In addition, four projects require a large amount of time for the experiment attendees to review all of them.

The first experiment uses the virtual machines in a CTF like event. The participants have to find the vulnerabilities. They can choose the method to find vulnerabilities on their own. They get points by submitting a vulnerability report, a patch to fix the vulnerability and by providing a working exploit for the discovered vulnerability. As the CTF event progresses, different hints are provided. For example, a hint that a specific input is potentially vulnerable. The Insecurity Refactoring process knows what source (a leaf of the ACID tree) has been used for the vulnerability. Accordingly, such hints can be provided easily to reduce the review process of the attendees.

In contrast, the second experiment is a guided exercise. The participants of that group get access to the virtual machines and the modified source code. They are guided to use static and dynamic analysis tools to find the vulnerabilities. At the end of the experiment, the participants get a list of all injected vulnerabilities to compare it to the results of the tools. The idea is to provide learning examples where the participants can learn the usage of static and dynamic tools. Accordingly, there has been inserted a data flow pattern that is difficult to be detected by dynamic tools and there have been added two vulnerabilities that are difficult to be detected by static code analysis tools. As the attendees get a list of all injected vulnerabilities, they can review the source to understand what source code patterns make it difficult for the tools to detect them. In this case, the pattern that is difficult for dynamic tools is a similar pattern to the pattern that LAVA and EvilCoder uses. It requires a specific input to disable a sanitization method. A dynamic tool would require to guess that input, that would require a fuzzy methodology or to brute force it. In contrast, a static code analysis tools does not require guessing the input as it can be read in the source code.

The virtual machines and the vulnerabilities are different for both experiments. The usage of the virtual machines shows if the normal usage of the program is still working. Any reports from the participants that something is not working correctly would show that the Insecurity Refactoring definition is not held. To evaluate the usage of learning examples, a survey is the choice of method. A survey before the events checks the skill level of the participants. The survey after the event verifies if the exercises created by Insecurity Refactoring has been useful for the participants.

As the tools LAVA and EvilCoder are similar to the Insecurity Refactoring approach, functional and experimental comparisons have been done. The

functional comparison compares how they detect possibilities for injection, how the vulnerabilities injected, and how realistic the injected vulnerabilities are. The experimental comparison is more difficult because LAVA is for Java projects and EvilCoder is for C projects. Accordingly, a comparison on the same database is not possible. For the experimental comparison, only the EvilCoder tool has been compared to our approach because LAVA uses a dynamic approach that is very different than the static approach of EvilCoder and Insecurity Refactoring. The evaluation on open-source projects from EvilCoder is similar to our evaluation. For the comparison, the evaluation of EvilCoder on open-source projects have been compared to our results of the evaluation on open-source projects. Similar metrics have been calculated and compared to each other.

5.5 M5: Benchmark static code analysis tools

Previous work provides different source code patterns represented in PL/V language. In addition, the Insecurity Refactoring process also introduced the functionality to create source code based on those patterns. The idea was to create test suites that contain vulnerable source code based on our identified patterns. Figure 5.6 shows the method that is used to create such test suites. A review of state-of-the-art benchmark data sets show that for PHP, the

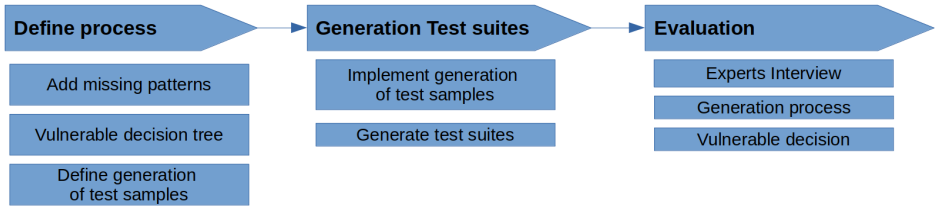


Figure 5.6: Test suite for static code analysis tools.

newest data set is the PHP Vulnerability Test Suite [74]. The review showed that those test suites are also created by combining different source code parts together. They used it in a more simple way that there are source code snippets for sources, sanitization functions and sinks. Nevertheless, a comparison of the patterns they use shows if the Insecurity Refactoring framework misses any patterns. The missing patterns are added to have a complete set of source code patterns as the static code analysis benchmark set. All the obtained source code patterns except data flow patterns are used to create the first test suite that contains all possible combinations of the

patterns. For the data flow patterns, a second test suite is generated, where one sample represents one data flow pattern. Per definition, our data flow patterns do not change the data itself. Accordingly, it would just multiply the number of test cases of the first test suite by the number of data flow patterns.

The source code itself would be sufficient to create a test suite. Nevertheless, there is a problem that some sanitization functions are filtering only specific characters. The previous implementation for Insecurity Refactoring decided based on sanitization function, context and sink if the sanitization function is sufficient or insufficient. We improved our previous definition if a vulnerability is vulnerable. A decision tree has been introduced to determine whether a test case is vulnerable or not. For this decision tree, it has been defined what characters are filtered out by sanitization functions. Different decision trees are defined for SQL Injection and Cross Site Scripting. Those decision trees define based on the context and what characters are allowed if a sample is vulnerable.

In addition, we improved the generation process that each sample contains a docker file. This allows to easily verify if the generated samples are runnable and check if a vulnerable sample is actually vulnerable. Based on the improved decision tree, it even allows providing hints what characters can be used to exploit the vulnerability. A docker file must be very specific, especially for the SQL databases. Each database driver requires a running database in the background. For each pattern, it is defined what is required in the docker file. Based on those requirements, the docker files can be generated. Then the Docker files have been used to verify whether the samples are actually working or not.

The contribution of these samples aims to benchmark static code analysis tools. As the commercial tools performed better in our previous work, we decided to use only commercial static code analysis tools to scan all the generated samples. To evaluate if these samples provide a solid benchmark data set, the results from the static code analysis tools are used to calculate already established static code analysis metrics. The metrics show if the data set can be used to identify problems of the static code analysis tools. An experts interview with benchmark experts from SAMATE has been used to show whether the approach generating these samples is useful or not. Especially, the decision tree is a central point of the generation process. The expert interview has been done to point out issues of the generation process. The feedback from the interview has been used to improve the test suites.

Chapter 6

List of published papers

This section provides an overview of the published papers.

- **Paper A: Source Code Patterns of SQL Injection Vulnerabilities**
  Felix Schuckert, Basel Katt, Hanno Langweg
  Conference: ARES 2017 - 12th International Conference on Availability, Reliability and Security
  Abstract: Many secure software development methods and tools are well-known and understood. Still the same software security vulnerabilities keep occurring. To find out if new source code patterns evolved or the same patterns are reoccurring, we investigate SQL Injections in PHP open source projects. SQL Injections are well-known and a core part of software security education. For each common part of SQL Injections the source code patterns are analyzed. Examples are pointed out showing that developers had software security in mind, but nevertheless created vulnerabilities. Our main contribution is the categorization of source code patterns.

- **Paper B: Source Code Patterns of Buffer Overflow Vulnerabilities in Firefox**
  Felix Schuckert, Max Hildner, Hanno Langweg, Basel Katt
  Conference: Proceedings of Sicherheit 2018
  Abstract: We investigated 50 randomly selected buffer overflow vulnerabilities in Firefox. The source code of these vulnerabilities and the corresponding patches were manually reviewed and patterns were identified. Our main contribution are taxonomies of errors, sinks and fixes seen from a developer's point of view. The results are compared to the CWE taxonomy with an emphasis on vulnerability details. Ad-

ditionally, some ideas are presented on how the taxonomy could be used to improve the software security education.

- **Paper C: Source Code Patterns of Cross Site Scripting in PHP Open Source Projects**
  Felix Schuckert, Max Hildner, Basel Katt, Hanno Langweg
  Conference: Proceedings of the 11th Norwegian Information Security Conference
  Abstract: To get a better understanding of Cross Site Scripting vulnerabilities, we investigated 50 randomly selected CVE reports which are related to open source projects. The vulnerable and patched source code was manually reviewed to find out what kind of source code patterns were used. Source code pattern categories were found for sources, concatenations, sinks, html context and fixes. Our resulting categories are compared to categories from CWE. A source code sample which might have led developers to believe that the data was already sanitized is described in detail. For the different html context categories, the necessary Cross Site Scripting prevention mechanisms are described.

- **Paper D: Difficult XSS Code Patterns for Static Code Analysis Tools**
  Felix Schuckert, Basel Katt, Hanno Langweg
  Conference: 1st Model-Driven Simulation and Training Environment for Cybersecurity
  Abstract: We present source code patterns that are difficult for modern static code analysis tools. Our study comprises 50 different open source projects in both a vulnerable and a fixed version for XSS vulnerabilities reported with CVE IDs over a period of seven years. We used three commercial and two open source static code analysis tools. Based on the reported vulnerabilities we discovered code patterns that appear to be difficult to classify by static analysis. The re- sults show that code analysis tools are helpful, but still have problems with specific source code patterns. These patterns should be a focus in training for developers.

- **Paper E: Difficult SQLi Code Patterns for Static Code Analysis Tools**
  Felix Schuckert, Basel Katt, Hanno Langweg
  Conference: Norsk IKT-konferanse for forskning og utdanning. No. 3. 2020
  Abstract: We compared vulnerable and fixed versions of the source code

of 50 different PHP open source projects based on CVE reports for SQL Injection vulnerabilities. We scanned the source code with commercial and open source tools for static code analysis. Our results show that five current state-of-the-art tools have issues correctly marking vulnerable and safe code. We identify 25 code patterns that are not detected as a vulnerability by at least one of the tools and 6 code patterns that are mistakenly reported as a vulnerability that cannot be confirmed by manual code inspection. Knowledge of the patterns could help vendors of static code analysis tools, and software developers could be instructed to avoid patterns that confuse automated tools.

- **Paper F: Insecurity Refactoring**
  Felix Schuckert, Basel Katt, Hanno Langweg
  Journal: Computer & Security, Volume 128, 2023
  Received 30 April 2021, Revised: 21 December 2022, Accepted: 24 January 2023
  Abstract:Insecurity Refactoring is a change to the internal structure of software to inject a vulnerability without changing the observable behavior in a normal use case scenario. An implementation of Insecurity Refactoring is formally explained to inject vulnerabilities in source code projects by using static code analysis. It creates learning examples with source code patterns from known vulnerabilities.

  Insecurity Refactoring is achieved by creating an Adversary Controlled Input Dataflow tree based on a Code Property Graph. The tree is used to find possible injection paths. Transformation of the possible injection paths allows to inject vulnerabilities. Insertion of data flow patterns introduces different code patterns from related Common Vulnerabilities and Exposures (CVE) reports. The approach is evaluated on 307 open source projects. Additionally, insecurity-refactored projects are deployed in virtual machines to be used as learning examples. Different static code analysis tools, dynamic tools and manual inspections are used with modified projects to confirm the presence of vulnerabilities.

  The results show that in 8.1% of the open source projects it is possible to inject vulnerabilities. Different inspected code patterns from CVE reports can be inserted using corresponding data flow patterns. Furthermore the results reveal that the injected vulnerabilities are useful for a small sample size of attendees (n=16). Insecurity Refactoring is useful to automatically generate learning examples to improve software security training. It uses real projects as base whereas the injected vulnerabilities stem from real CVE reports. This makes the injected

vulnerabilities unique and realistic.

- **Paper G: Systematic Generation of XSS and SQLi Vulnerabilities in PHP as Test Cases for Static Code Analysis** *Felix Schuckert, Basel Katt, Hanno Langweg*
  Conference: 2022 IEEE International Conference on Software Testing, Verification and Validation Workshops
  Abstract: Synthetic static code analysis test suites are important to test the basic functionality of tools. We present a framework that uses different source code patterns to generate Cross Site Scripting and SQL Injection test cases. A decision tree is used to determine if the test cases are vulnerable. The test cases are split into two test suites. The first test suite contains 258,432 test cases that have influence on the decision trees. The second test suite contains 20 vulnerable test cases with different data flow patterns. The test cases are scanned with two commercial static code analysis tools to show that they can be used to benchmark and identify problems of static code analysis tools. Expert interviews confirm that the decision tree is a solid way to determine the vulnerable test cases and that the test suites are relevant.

Chapter 7

Contribution

This section describes the contribution of this work. The choice of methods for each scientific question are already described. Figure 7.1 shows an overview

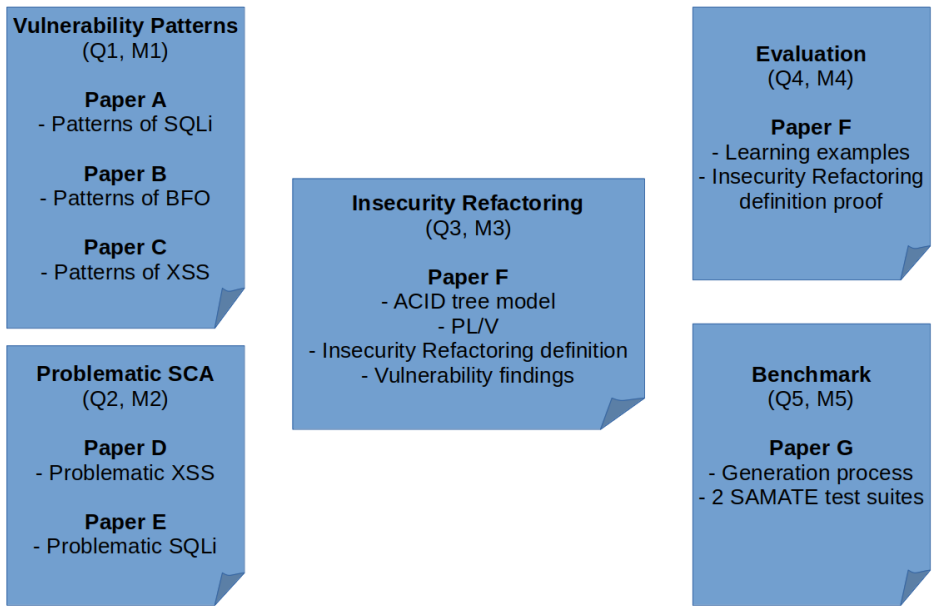


Figure 7.1: Contribution overview.

of what papers contributed to the research questions. The following section describes what each paper contributes to the corresponding question in detail. Additionally, any critics and thoughts for each paper are discussed.

7.1 Source Code Patterns Classification of Vulnerabilities

Paper A, B and C have followed the same methodology M1 but with different data sets. A crawler has been developed to retrieve the data sets. The crawler looked up for CVE reports that are related to open source projects. The results have indicated that most vulnerabilities are found for the programming languages PHP and C/C++. These results directed the research into SQL Injection in PHP (paper A), Buffer Overflow in C/C++ (paper B) and Cross Site Scripting in PHP (paper C).

Paper A has focused on SQL Injection vulnerabilities in PHP open source projects. The source code patterns have been classified into sources, concatenations, failed sanitizations, sinks and fixes. Compared to the classification of Shar et al. [73], our categories are focused on the source code. For example, their sinks are just categorized into SQL Injection and Cross Site Scripting. Our results are sub categories for SQL Injection sinks.

It has shown that many of the vulnerabilities did not include any sanitization methods. Some samples were just plain SQL Injection samples as found in teaching examples. A special case has shown that developers used the *header* function to redirect the user if unwanted inputs are found. But they probably did not know that the source code will be executed after the redirect. The official PHP API description misses a critical part and only states in a comment that the *exit* function should be called after using the redirect. The paper A has contributed a detailed classification of SQL Injection vulnerabilities in PHP and how developers fixed the vulnerabilities.

Paper B has provided a classification of Buffer Overflow vulnerabilities found in Firefox. Buffer Overflow vulnerabilities are still a vulnerability type that occurs frequently. Buffer Overflow (CWE-119) is still in the fifth place in the 2020 CWE Top 25 [9]. Firefox has been used as a data set because they use the Bugzilla [31] platform to track bugs and vulnerabilities. For the vulnerabilities, the developers have commented on how they fixed the vulnerabilities. The classification has been created for type of error, sink and how the vulnerabilities are patched. Additionally, the type of error has been classified on what type of error the developers did. The results have shown that the developers tried to prevent Buffer Overflow vulnerabilities. Only a few samples (5/50) had no prevention attempts. Most vulnerabilities occurred because of some Integer variable overflow. An overflowed variable in a combination with size checks has provided unexpected results resulting in a Buffer Overflow vulnerability. Furthermore, unexpected inputs like negative numbers were problematic for size checks. Additionally, Firefox is written

in the programming language C/C++. The programming language C/C++ allows assigning different variables types to each other. In combination with size checks, this can be problematic. For example, six vulnerabilities had an assignment of an Unsigned Integer to a Signed Integer. This is not problematic as long as the first bit of the Unsigned Integer is zero. For a large number, where the first bit is not zero, the assigned variable will have a negative number. Again, in combination with a size check, this resulted in Buffer Overflow vulnerabilities.

The sinks have been classified into critical functions, arrays, and pointers. Nothing special has been found in the data set. The results show that most had critical functions as sink and the least samples had pointers as sink.

The fixes of the developers have been classified. The results show that the developers mainly resolved the issues based on the type of error they did. For example, if an integer variable can overflow, they added checks to prevent that overflow. In addition, the variable types have been changed to the correct type. For example, an integer has been changed to an unsigned integer to prevent the issues of assigning different data types.

One special case was interesting, where a list of entries was sanitized in order. But a situation can occur that the ordering of the list not as expected. The order result that the sanitization of some entries of the list was skipped and resulted in a buffer overflow vulnerability.

Overall, the paper B provides an overview where the data set is narrow. It only uses vulnerabilities of one project. The results cannot be considered as a taxonomy of all buffer overflow vulnerabilities. The developers of Firefox might have a higher skill level and better education than the developers of random selected open source projects from GitHub [19].

Paper C has provided a detailed classification of Cross Site Scripting vulnerabilities in PHP open source projects. Again, the results have shown that many vulnerabilities did not include any sanitization methods. Another contribution of that paper has shown that the different sanitization methods depend heavily on the context of the user input. Depending on the context, a sanitization might be sufficient to prevent a vulnerability or not. This makes it difficult for software developers to know which sanitization methods are sufficient. The results have shown that some sanitization functions do not filter all critical characters by default. In paper C, we mentioned that the default filter should filter all critical characters and developers should make a conscious decision to allow critical characters. If so, developers don't have to know what special characters are filtered by specific sanitization functions. Instead, by specifying what characters are passed, they know that these special characters are allowed. The classifications have been mapped

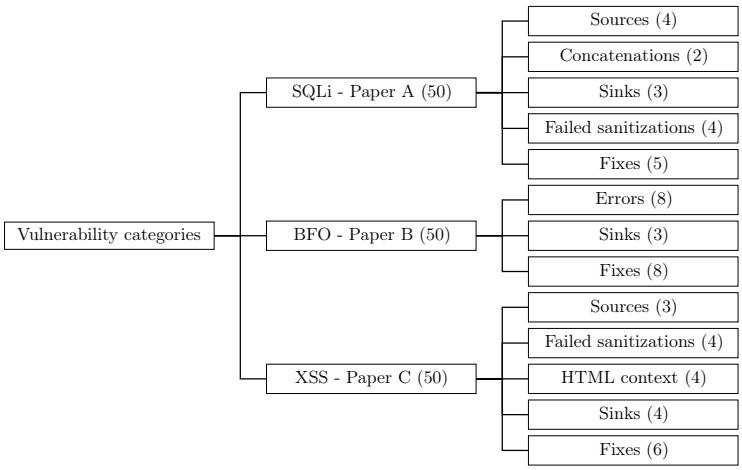


Figure 7.2: The amount of source code pattern categories identified from 150 vulnerabilities.

to known CWE categories. Not all of them have been mapped to CWE categories. Some of our categories are more detailed. For example, CWE has the html attribute *img* as CWE-82. We also found other html attributes e.g., *input*, *div* that don't have a CWE category. For other categories like the sink, the perspective of a developer provides other categories than the CWE categories. The CWE categories are more based on the vulnerability itself and not the underlying source code.

The three papers A, B and C have provided a classification of three different vulnerability types (SQL Injection, Cross Site Scripting and Buffer Overflow). Figure 7.2 shows how many categories have been identified based on the reviewed vulnerabilities. The categories from SQLi and XSS can be unified into 5 sources, 2 concatenations, 6 sinks, 7 failed sanitization, 4 HTML context and 11 fixes categories. Accordingly, the three papers classified the source code from 150 vulnerabilities into 54 unique categories. The data set of 150 entries has not been large enough to ensure a full classification of the vulnerability types. Because it has been required to review the vulnerable and fixed versions. The manual reviewing has required significant effort. This limits the number of entries that can be reviewed. But the results from papers D and F show that state-of-the-art static code analysis tools cannot provide precise enough results to create a classification.

7.2 Difficult source code patterns for static code analysis tools

Paper D and E contribute to publish source code patterns that are problematic for state-of-the-art static code analysis tools. The same data sets from paper A and paper C have been used to find problematic source code patterns. The commercial static code analysis tools are not named. We reference them here as tool A, tool B and tool C. Those references are the same in the published papers.

First of all, paper D has looked into Cross Site Scripting vulnerabilities. The initial scan has shown that the commercial tools were able to detect over 50% of the vulnerabilities. Accordingly, the results have shown that over half of the vulnerabilities that occurred could have been prevented if these static code analysis tools had been used on the open-source projects. The process of identifying the critical source code patterns and the evaluation of them has revealed 23 source code patterns that are problematic to be detected by modern static code analysis tools. The source code patterns have been published in combination with paper D. The results have shown that commercial static code analysis tools perform better than open source static code analysis tools on the data set. For a stored cross site scripting vulnerability, the critical user data is stored in the database and displayed on the website later on. Figure 7.3 displays an overview of a stored Cross Site Scripting sample. The relation between the select and insert makes it difficult for static code analysis tools to correctly detect a vulnerability. For a correct vulnerability detection, the relation between the insert/update SQL statement and the select SQL statement has to be resolved. Additionally, there are two possible places to prevent a stored Cross Site Scripting vulnerability. The user input can be sanitized before it is placed in the database. In addition, it can also be sanitized before the user data reaches the sink. It even might be sanitized in both cases at the same time. Source code samples have been created to represent the different scenarios. One tool (A) has been able to detect all stored Cross Site Scripting samples correctly. The other tools (B, C) will need improvements to correctly detect stored Cross Site Scripting vulnerabilities. These samples have shown that even basic stored Cross Site Scripting samples are problematic for some commercial static code analysis tools.

Paper E has used the SQL Injection data set to find source code patterns that are problematic for static code analysis tools. The initial scan has shown that the commercial tools have a higher detection rate on the SQL Injection sample than on the Cross Site Scripting samples from paper D. The overall

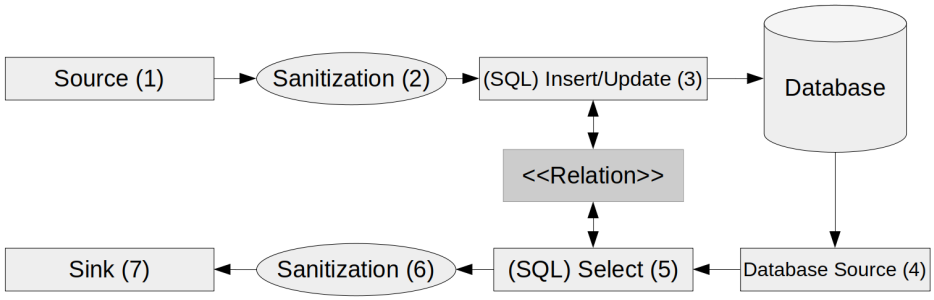


Figure 7.3: Overview of a stored XSS vulnerability [66].

process has revealed 31 source code pattern that are difficult for static code analysis tools. Many of these patterns are common PHP functionality, but still are difficult to detect. For example, all dynamic programming language features like dynamic function calls require additional steps for the static code analysis tools. The tools have to parse the dynamic function name that might even require additional data flow analysis to resolve the function name itself. Additionally, the results have shown that regular expressions are difficult for static code analysis tools. Sanitization based on regular expressions is sufficient based on the regular expression itself. Accordingly, the tools have to analyze the regular expression to determine if it is sufficient to prevent any attacks. Again, the results have shown that the commercial static code analysis tools perform better than the open source tools.

For both papers D and E, the corresponding samples that have been scanned are published on GitHub [13]. This allows static code analysis tool developers to improve their tools so that it can detect these difficult source code patterns. Additionally, these pattern can be taught to software developers. If developers teach these patterns and are using static code analysis tools in their development process, they know what source code patterns are critical. Such a critical source code pattern can either be prevented at the development stage or the developers know that the tools will not be able to track these patterns. The data set with 100 false positive and 100 false negative samples is still small. The identified source code pattern will not represent all problematic source code patterns. Nevertheless, it has revealed 54 source code patterns that are problematic for modern static code analysis tools. From these patterns, 44 are false negative samples and only 10 are false positive samples. The methodology has been focused on detecting false negative samples. A false positive sample can only occur, if the tools already detected the vulnerable sample correctly. Accordingly, if

a vulnerable sample is not detected by a tool, the patched sample cannot create a false positive report. To get more source code patterns that are problematic regarding false positive results, another methodology should be used.

7.3 Insecurity Refactoring

The main contribution is the Insecurity Refactoring part. Paper F has provided a process on how to inject vulnerabilities in existing projects. It has officially defined the Insecurity Refactoring as a term for injecting vulnerabilities.

From previous work, a lot of different source code patterns have been identified. The PL/V pattern language has been developed that is programming language independent. A pattern in PL/V is constructed by using different language patterns. A language pattern itself stores the abstract syntax tree representation for different programming languages. This decouples the identified source code pattern from specific programming languages.

The injection of a vulnerability requires finding a source code part that can be transformed into a vulnerability. In paper F, such a source code part is called a Possible Injection Path (PIP). The Insecurity Refactoring definition requires precise detection of PIPs. Paper F has introduced and has defined the rules to create an Adversary Controlled Input Dataflow (ACID) tree that is an ordered, rooted, directed, edge-labeled and attributed tree. It uses the Code Property Graph [78] and specific pattern types that are described in the PL/V language to create the ACID tree. The ACID tree is a new analysis model that represents a backwards data flow analysis. Each node in the tree represents if the children are a concatenation or excluding. This allows to detect the context where the user input will be used by combining all the concatenation of the tree structure. The ACID tree can be used to detect vulnerabilities. Vulnerability definition for an ACID tree have been defined. An evaluation on 307 open-source projects has shown that in 25 projects, PIPs have been found. Additionally, vulnerabilities have been detected and three CVE reports (CVE-2020-27163, CVE-2021-3318, CVE-2021-26716) have been created based on the findings. The other vulnerabilities that have been found were in non-critical parts (e.g., in a test file).

Figure 7.4 shows the process of Insecurity Refactoring. As described, the ACID tree has been used to find PIPs. Rules have been defined to inject vulnerabilities. It basically requires replacing all sanitization methods found in a PIP with sanitization methods that are insufficient in that context. An insufficient sanitization method can either be no sanitization or a sanitization

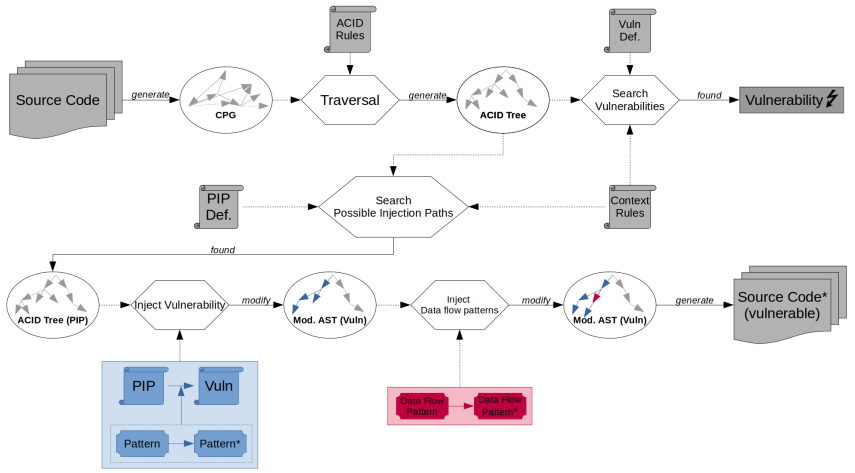


Figure 7.4: Overview of Insecurity Refactoring process by using an ACID tree. [67]

function that only filters specific characters but not all characters to prevent an attack. This would have been enough to inject a vulnerability. To get better learning examples, data flow patterns have been introduced. These are patterns where data flows through without actually transforming the data. Accordingly, they can be introduced without changing the behavior of the program. The data flow patterns have been used to implement the source code patterns which are problematic for static code analysis tools. Additionally, the special cases that have been found in paper A and paper C have been implemented as data flow patterns. The Insecurity Refactoring framework containing all the source code pattern written in PL/V has been published on GitHub [14]. This framework allows creating learning examples automatically. The insecurity refactoring evaluation process on 307 open source projects shows that it is possible to inject vulnerabilities in 35 of them. Accordingly, the tool could inject vulnerabilities in 8.1% of the projects. In addition, three vulnerabilities (CVE-2020-27163, CVE-2021-3318, CVE-2021-26716) have been detected in commonly used projects. A functional comparison with LAVA and EvilCoder shows that Insecurity Refactoring is more precise, with the disadvantage that it can inject fewer vulnerabilities. As our approach is based on the Insecurity Refactoring definition, it will not change the observable behavior in a normal use case scenario. LAVA and EvilCoder try to minimize that the changes to the normal use case scenario by injecting vulnerabilities in parts that are not used that often. But the goal of those tools was not maintaining the normal use case scenario.

In combination with the specific inputs that are required, those injected vulnerabilities provide a backdoor that has to be identified first. A security scanner that scans for critical characters would not trigger such specific inputs. Instead, a fuzzer or brute force tool would be able to detect such specific inputs. Such a vulnerability that requires a specific input would definitely not have been mistakenly written by developers. In contrast, the LAVA tools can create connections between critical sinks and user provided inputs does not ensure that the normal use case is ensured. The likelihood of changing the normal use case is minimized by using user input that is not often used in other parts of the program. As EvilCoder also uses the Code Property Graph, an experimental comparison with Insecurity Refactoring shows that EvilCoder finds more injection possibilities. In contrast, our approach allows to inject combinations of different source code patterns based on the possible injection path that has been found. For example, it allows to inject 5 different sources, 10 different data flow patterns and 9 different sanitization functions which result in $5 * 10 * 9 = 450$ different permutations.

7.4 Usage as learning examples

The goal of Insecurity Refactoring has been to create learning examples. Paper F has evaluated Insecurity Refactoring as learning examples by two experiments with different groups. The initial survey has shown that one group has been experienced in the field of information security and the other group has been relatively new. The events using the modified projects have shown that none injected vulnerabilities have created any problems in the normal usage of the programs. The post survey has shown that almost everyone had seen a skill increase at the event. Only two of the experienced attendees have not seen any skill increase. The results show that Insecurity Refactoring can be used to create learning examples.

The survey size (n=9+7=16) is critically small. Because of the pandemic situation, we could not run the events locally. Accordingly, the events have been run remotely. It has not been as easy to motivate and help the attendees in a remote situation because you could not see if they encountered any problems. In addition, the events have been optional for the attendees. This lowered the attendees of the survey size. The evaluation has shown that the insecurity refactored projects can be used in different learning exercises. Additionally, the combination of different insufficient sanitization methods and by adding data flow patterns has allowed to automatically create many different permutations of learning examples. The results were positive and have not revealed any problems with using the insecurity refactored projects

as learning examples.

7.5 Test suite for static code analysis tools

Paper G combines the results from all previously published papers. The different patterns from reviewed vulnerabilities combined with the critical source code patterns for static code analysis tools provide interesting test cases. Figure 7.5 shows the generation process of the test cases. The source, sanitization, context, and sink pattern stem from previous papers (A, B, C). The source code patterns that are difficult for static code analysis tools that stem from papers D and E are included as data flow patterns. Decision trees are introduced to decide if a test case is vulnerable or safe. The decision trees are a solid way to use the filtered characters and check for the given context and sink categories to decide if a test case is vulnerable. Six experts from the Software Assurance Metrics And Tool Evaluation (SAMATE) [5] at National Institute of Standards and Technology (NIST) [4] acknowledged that the test cases are useful and that the decision tree is solid. Two resulting test suites from paper G have been published at SAMATE as a Software Assurance Reference Data (SARD) set [7] [6]. One test suite contains all the different combinations of filtered characters, context, and sinks to cover all the possibilities in the decision trees. The other test suite contains all the difficult source code patterns for static code analysis tools. These test suites help the developers of static code analysis tools to detect the difficult patterns. Additionally, it can be used to make the tools more precise depending on the different contexts.

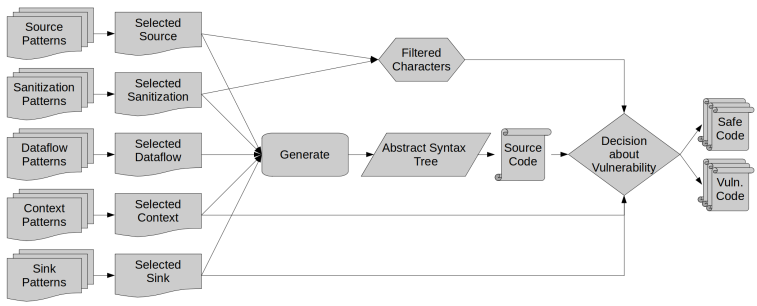


Figure 7.5: Test case generation process. [68]

Chapter 8

Future Work

The thesis provides a solid method to inject vulnerabilities in PHP source code. The current method only injects SQL Injection and Cross Site Scripting vulnerabilities. Both of those vulnerability types are based on a data flow from user input into a critical sink. Another question will be if the Insecurity Refactoring process can be extended to support other vulnerabilities. Especially for vulnerability types that don't have a specific sink. For example, can the Insecurity Refactoring process be extended to inject a *Security Misconfiguration* or a *Broken Access Control* vulnerability? In addition, it would also be interesting to increase the programming language support. The PL/V language has been defined in a way that it is programming language independent. Nevertheless, many patterns that stem from PHP are still specific to the programming language. For example, *htmlspecialchars* is a programming language-specific sanitization function. In the future, it might be interesting to inject vulnerabilities in different programming languages. Especially because some vulnerability categories like *Buffer Overflow* are not possible in PHP.

Another interesting question would be if the source code patterns of the vulnerabilities have changed. Newer vulnerabilities could be reviewed if the developers have made the same mistakes as in our research. Reoccurring patterns are interesting because those are probably parts that should be taught to software developers. New patterns are also interesting to see what has changed and those patterns could be extended to the Insecurity Refactoring process.

The thesis evaluated that Insecurity Refactoring can be used as learning examples. Nevertheless, it misses an evaluation on how effective it is. A comparative analysis to other commonly used learning examples would be helpful. In addition, it could also be evaluated how useful Insecurity Refactoring is

as repetitive learning examples. It allows to inject vulnerabilities in different projects and it can inject different vulnerabilities based on the injected source code patterns. It would be interesting to evaluate if the resulting learning examples will be unique enough as repetitive learning examples. It would also be interesting if it is possible to categorize the injected vulnerabilities regarding difficulty levels. An idea would be based on the code base, the ACID tree length and the injected patterns that a difficulty for the injected vulnerability could be estimated. This would require a methodology to determine the difficulty. An evaluation could show if the determined difficulties are useful and precise.

In the last few years, AI-based text generation like Chat Generative Pre-trained Transformer (ChatGPT) have increased in popularity. It would be interesting to see if a AI-based creation or injection of vulnerabilities would be possible. It can be checked, if it is possible to inject a vulnerability without breaking the functionality of the program. In addition, another interesting question would be to compare such a methodology to Insecurity Refactoring. Again, a comparative evaluation between those methods would be interesting.

References

[1] CVE Common Vulnerabilities and Exposures. https://cve.mitre.org/, .

[2] CVE-2001-1471. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-1471, .

[3] Juliet Test Suite. http://samate.nist.gov/SRD/testsuite.php.

[4] National Institute of Standards and Technology. https://www.nist.gov/.

[5] SAMATE. https://www.nist.gov/itl/ssd/software-quality-group/samate.

[6] Test Suite - Data flow. https://samate.nist.gov/SARD/test-suites/115, .

[7] Test Suite - XSS/SQLi. https://samate.nist.gov/SARD/test-suites/114, .

[8] WebGoat GitHub. https://github.com/WebGoat/WebGoat/releases.

[9] Common Weakness Enumeration. http://cwe.mitre.org/, 2015.

[10] OWASP Top Ten Project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2015.

[11] Constructing benchmarks for supporting explainable evaluations of static application security testing tools. *Proceedings - 2019 13th International Symposium on Theoretical Aspects of Software Engineering, TASE 2019*, pages 65–72, 2019.

[12] Clang. https://clang.llvm.org/, 2021.

[13] Insecurity Refactoring. https://github.com/fschuckert/sca_patterns, 2021.

[14] Insecurity Refactoring. https://github.com/fschuckert/insecurity-refactoring, 2021.

[15] OWASP Static Code Analysis. https://www.owasp.org, 2021.

[16] OWASP top 10 - 2017. https://owasp.org/www-project-top-ten/, 2021.

[17] PHP repository - backdoor commit. https://github.com/php/php-src/commit/c730aa26bd52829a49f2ad284b181b7e82a68d7d, 2021.

[18] rr debugger. https://rr-project.org/, 2021.

[19] GitHub. https://github.com/, 2022.

[20] InfiniteGraph. https://infinitegraph.com/, 2022.

[21] neo4j graph data platform. https://neo4j.com/, 2022.

[22] H. H. AlBreiki and Q. H. Mahmoud. Evaluation of static analysis tools for software security. *2014 10th International Conference on Innovations in Information Technology (IIT)*, pages 93–98, 2014. doi: 10.1109/INNOVATIONS.2014.6987569.

[23] F. E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, page 1–19, New York, NY, USA, 1970. Association for Computing Machinery. ISBN 9781450373869. doi: 10.1145/800028.808479.

[24] N. Antunes and M. Vieira. Comparing the effectiveness of penetration testing and static code analysis on the detection of SQL injection vulnerabilities in web services. *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2009*, pages 301–306, 2009. doi: 10.1109/PRDC.2009.54.

[25] T. Aslam. A taxonomy of security faults in the unix operating system. *Master's thesis, Purdue University*, 199(5), 1995.

[26] T. Aslam, I. Krsul, and E. H. Spafford. Use of a taxonomy of security faults. 1996.

[27] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and Flexible Discovery of PHP Application Vulnerabilities. *Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017*, pages 334–349, 2017. doi: 10.1109/EuroSP.2017.14.

[28] M. Bishop, S. Engle, D. Howard, and S. Whalen. A taxonomy of buffer overflow characteristics. *IEEE Transactions on Dependable and Secure Computing*, 9(3):305–317, 2012. ISSN 15455971. doi: 10.1109/TDSC.2012.10.

[29] T. Boland and P. E. Black. Juliet 1.1 C/C++ and Java Test Suite. *Computer*, 45(10):88–90, oct 2012. ISSN 1558-0814. doi: 10.1109/MC.2012.345.

[30] J. W. Brian Chess. *Secure Programming with Static Analysis*. 2007. ISBN 978-0321424778.

[31] Bugzilla, 2017. URL `https://bugzilla.mozilla.org`.

[32] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering*, 2021.

[33] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., Aug. 2003. USENIX Association. URL `https://www.usenix.org/conference/12th-usenix-security-symposium/static-analysis-executables-detect-malicious-patterns`.

[34] K. D. Cooper, T. J. Harvey, and T. Waterman. Building a control-flow graph from scheduled assembly code. Technical report, 2002.

[35] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. LAVA: Large-Scale Automated Vulnerability Addition. *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, pages 110–121, 2016. doi: 10.1109/SP.2016.15.

[36] K. Erno. Sticking to the facts II: scientific study of static analysis tools. *in: Presentation at the SATE IV Workshop, Center for Assured Software, National Security Agency*, 2012.

[37] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002. doi: 10.1109/52.976940.

[38] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9 (3):319–349, July 1987. ISSN 0164-0925. doi: 10.1145/24039.24041.

[39] S. M. Ghaffarian and H. R. Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):1–36, 2017.

[40] K. Goseva-Popstojanova and A. Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, 2015. ISSN 09505849.

[41] M. Howard and S. Lipner. The Security Development Lifecycle. *Change*, 34(May):352, 2006. ISSN 16140702. doi: 10.1007/s11623-010-0021-7.

[42] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 611–620, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605588940. doi: 10.1145/1653662.1653736. URL `https://doi.org/10.1145/1653662.1653736`.

[43] Z. Hui, S. Huang, B. Hu, and Z. Ren. A taxonomy of software security defects for SST. *Proceedings - 2010 International Conference on Intelligent Computing and Integrated Systems, ICISS2010*, pages 99–103, 2010. doi: 10.1109/ICISS.2010.5656736.

[44] Z. Hui, S. Huang, B. Hu, and Y. Yao. Software security testing based on typical SSD: A case study. *ICACTE 2010 - 2010 3rd International Conference on Advanced Computer Theory and Engineering, Proceedings*, 2:312–316, 2010. doi: 10.1109/ICACTE.2010.5579101.

[45] P. Hulin, A. Davis, R. Sridhar, A. Fasano, C. Gallagher, A. Sedlacek, T. Leek, and B. Dolan-Gavitt. Autoctf: Creating diverse pwnables via automated bug injection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, Aug. 2017. USENIX Association.

[46] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys (CSUR)*, 26(3):211–254, 1994.

[47] P. Lerthathairat and N. Prompoon. An approach for source code classification to enhance maintainability. *Proceedings of the 2011 8th International Joint Conference on Computer Science and Software Engineering, JCSSE 2011*, pages 319–324, 2011. doi: 10.1109/JCSSE.2011.5930141.

[48] X. Li, X. Chang, J. A. Board, and K. S. Trivedi. A novel approach for software vulnerability classification. Institute of Electrical and Electronics Engineers Inc., 3 2017. ISBN 9781509052844. doi: 10.1109/RAM.2017.7889792.

[49] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020.

[50] B. Livshits and S. Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, page 385–398, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318327. doi: 10.1145/2429069.2429115. URL `https://doi.org/10.1145/2429069.2429115`.

[51] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, July 2005. USENIX Association. URL `https://www.usenix.org/conference/14th-usenix-security-symposium/finding-security-vulnerabilities-java-applications-static`.

[52] K. Maruyama and T. Omori. A Security-Aware Refactoring Tool for Java Programs. *Proceedings - International Conference on Software Engineering*, pages 22–28, 2011. ISSN 02705257. doi: 10.1145/1984732.1984737.

[53] I. Medeiros, N. Neves, and M. Correia. Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining. *IEEE Transactions on Reliability*, 65(1):54–69, 2016.

[54] A. Møller and M. I. Schwartzbach. Static program analysis. *Notes. Feb*, 2012.

[55] L. Moonen. Generating robust parsers using island grammars. In *Proceedings eighth working conference on reverse engineering*, pages 13–22. IEEE, 2001.

[56] M. H. Nguyen, T. B. Nguyen, T. T. Quan, and M. Ogawa. A hybrid approach for control flow graph construction from binary code. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 2, pages 159–164, 2013. doi: 10.1109/APSEC.2013.132.

[57] M. H. Nguyen, T. B. Nguyen, T. T. Quan, and M. Ogawa. A hybrid approach for control flow graph construction from binary code. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 2, pages 159–164. IEEE, 2013.

[58] P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia, and M. Vieira. Benchmarking Static Analysis Tools for Web Security. *IEEE Transactions on Reliability*, 67(3):1159–1175, 2018. ISSN 00189529. doi: 10.1109/TR.2018.2839339.

[59] P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia, and M. Vieira. An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios. *Computing*, 101 (2):161–185, 2019. ISSN 0010485X. doi: 10.1007/s00607-018-0664-z.

[60] C. Oliveira. Real world software assurance test suite: Stonesoup. In *Proceedings of Software Technology Conference (STC), October*, pages 13–15, 2015.

[61] J. Pewny and T. Holz. Evilcoder: automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 214–225, 2016.

[62] PHP Documentation. `https://www.php.net/manual/`, 2023.

[63] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 49–61, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897916921. doi: 10.1145/199448.199462. URL `https://doi.org/10.1145/199448.199462`.

[64] M. A. Rodriguez and P. Neubauer. The graph traversal pattern. In *Graph data management: Techniques and applications*, pages 29–46. IGI Global, 2012.

[65] F. Schuckert, M. Hildner, B. Katt, and H. Langweb. Source code patterns of cross site scripting in php open source projects. *NISK Journal*, 11, 2018.

[66] F. Schuckert, B. Katt, and H. Langweg. Difficult XSS Code Patterns for Static Code Analysis Tools. In *Computer Security*, pages 123–139, Cham, 2020. Springer International Publishing. ISBN 978-3-030-42051-2. doi: 10.1007/978-3-030-42051-2_9.

[67] F. Schuckert, B. Katt, and H. Langweg. Insecurity refactoring. *Submitted to: Computers and Security*, 2021.

[68] F. Schuckert, B. Katt, and H. Langweg. Generation of php vulnerability test cases for xss and sqli. *Submitted to: 15th IEEE International Conference on Software Testing, Verification and Validation*, 2022.

[69] S. Shang, N. Zheng, J. Xu, M. Xu, and H. Zhang. Detecting malware variants via function-call graph similarity. In *2010 5th International Conference on Malicious and Unwanted Software*, pages 113–120, 2010. doi: 10.1109/MALWARE.2010.5665787.

[70] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium (USENIX Security 01)*, Washington, D.C., Aug. 2001. USENIX Association. URL `https://www.usenix.org/conference/10th-usenix-security-symposium/detecting-format-string-vulnerabilities-type-qualifiers`.

[71] L. K. Shar and H. B. K. Tan. Predicting common web application vulnerabilities from input validation and sanitization code patterns. *2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 - Proceedings*, pages 310–313, 2012. doi: 10.1145/2351676.2351733.

[72] L. K. Shar and H. K. Tan. Mining Input Sanitization Patterns for Predicting SQL Injection and Cross Site Scripting Vulnerabilities. pages 1293–1296, 2012. ISBN 9781467310673.

[73] L. K. Shar, H. Beng Kuan Tan, and L. C. Briand. Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis. *Proceedings - International Conference on Software Engineering*, pages 642–651, 2013. ISSN 02705257. doi: 10.1109/ICSE.2013.6606610.

[74] B. Stivalet and E. Fong. Large Scale Generation of Complex and Faulty PHP Test Cases. *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, pages 409–415, 2016. doi: 10.1109/ICST.2016.43.

[75] S. Thomas, L. Williams, and T. Xie. On automated prepared statement generation to remove SQL injection vulnerabilities. *Information and Software Technology*, 51(3):589–598, 2009. ISSN 09505849. doi: 10.1016/j.infsof.2008.08.002.

[76] A. L. S. Ullman. *Compilers Principles, Techniques, and Tools Second Edition*. 2013. ISBN 978-1292024349.

[77] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981. ISBN 0897911466.

[78] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. *Proceedings - IEEE Symposium on Security and Privacy*, pages 590–604, 2014. ISSN 10816011. doi: 10.1109/SP.2014.44.

[79] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[80] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *ACM SIGSOFT Software Engineering Notes*, (6):97. ISSN 01635948. doi: 10.1145/1041685.1029911.

Part I

Published Papers

Chapter 9

Source Code Patterns of SQL Injection Vulnerabilities

9.1 Introduction

Looking into the OWASP [86] Top Ten shows that the same vulnerability categories are occurring all the time. One of the top categories is still *Injection*. A lot of research has been done on SQL injection. To discover the reason why the same vulnerabilities are occurring, we investigated the source code from open source projects. For the source code, similar methods and operations are grouped up and are called source code patterns. Our work shows which source code patterns occur in real life projects, to provide a data set that can be compared to existing vulnerability data sets like SAMATE SARD [91]. Another aspect will be using this data set to provide exercises for software developers to learn or improve their software security skills. Training developers with a data set based on existing vulnerabilities helps developers to identify vulnerabilities beyond artificial samples. The first question to be answered is: How does such source code patterns look like? The next question is, are there any special cases that are not typical for SQL injection vulnerabilities? Real source code samples are investigated to get answers to these questions.

In this research we use a crawler to get source code from open source projects.

Reviewed are entries that are listed in the Common Vulnerabilities and Exposures (CVE) database, which belong to the Injection category and were related to projects using PHP as programming language. We choose PHP because it is still a popular programming language. Additionally, many vulnerabilities in various categories are possible in PHP. We categorize the

manual review results in source code patterns. This research provides an overview of the kind of source code patterns that occurred within the last six years of SQL injections.

This paper begins with an overview of related works in section 9.2. The next section explains how the source code was obtained. Additionally, the section explains the review method. The results from the crawler and manual review are shown in section 9.4. The found source code patterns are categorized. Some special cases are explained in detail with the corresponding source code. The last two sections provide a discussion about the results and suggestions for future work.

9.2 Previous and related work

Classifications of security vulnerabilities from the Open Web Application Security Project (OWASP) [86] and Common Weakness Enumeration (CWE) [84] already exist. The first classification research projects of security in operating systems and programs did already occur in the 1970s. Abbott et al. [92] introduced a taxonomy for security flaws in operating systems. Another article from Bisbey and Hollingworth [94] did an error categorization. Both research projects analysed security flaws in operating systems. At that time, no security vulnerabilities like SQL injection existed. Another approach from Seacord and Householder [100] classifies security flaws using properties instead of a taxonomy. It assigns properties to source code parts and based on these properties, security flaws are classified. Furthermore, a technique is defined where these assigned properties can be matched to properties from security flaws that have already been classified. Thus, successfully matched properties can classified into known categories.

Research projects about general classification of source code patterns exist. Lerthathairat and Prompoon [96] did research about the classification of source code into bad code, clean code and ambiguous code. They use metrics in source code like comments, the size of the function, et cetera. These metrics will be analysed by using Fuzzy logic to determine which category the source code belongs to. Bad and ambiguous code will be improved by refactoring. A more security related work is from Hui et al. [95], where a software security taxonomy for software security tests is used. They created a security defects taxonomy based on top 10 software security errors from authoritative vulnerability enumerations. Their taxonomy is categorized into *induced causes*, *modification methods* and *reverse use methods*. They advice that their taxonomy should be used as security test cases. Stivalet and Fong [103] present a tool that allows to create short code examples

including software vulnerabilities. They split up the source code parts into *input*, *filtering* and *sink*. Permutations of these parts will create different examples. All of the examples can be found in the *testsuite 103* within the Software Assurance Reference Dataset [91].

A related work more specific to SQL injection is from Shar and Tan [101]. They predict the probability of SQL injections or cross site scripting based on sanitization patterns by using a data flow analysis for the prediction. They classified sources, sanitization methods and sinks. The successive paper from Shar et al. [102] did classify the sanitization methods more precisely. This was archieved through a hybrid program analysis where dynamic and static code analysis was combined in order to find SQL injection and cross site scripting vulnerabilities. Both have similar categories as this research. Nevertheless, the categories are not based on an existing source code data set. A comparison of categories is discussed in section 9.5. Medeiros et al. [99] explain the *WAP* tool. It uses static code analysis and machine learning to detect vulnerabilities. They did a manual review of source code that was detected by their tool in order to verify true positives. In the work their categories for sanitization methods are presented.

The research also requires data sources which are used to classify the source code patterns. Massacci and Nguyen [97] researched different data sources for vulnerabilities, e.g. CVE, NVD, et cetera. They checked which data sources were used by other research projects. The result showed that some projects missed a large portion of vulnerabilities because they only analysed Firefox. Wu et al. [104] use semantic templates that are created from the CWE database [84]. These templates should help to understand security vulnerabilities. The authors did an empirical study to prove that these semantic templates have a positive impact on the time to completion on finding the vulnerability.

9.3 Method

Figure 9.1 shows an overview of the method used to determine the source code patterns. *CVE Details* [83] categorizes vulnerability types for each CVE entry. The classification is based on matching keywords and searching for CWE numbers. It is stated that these categories may not be reliable. These faulty categorized CVE entries are detected in the manual review process and are ignored to determine source code patterns. The functionality of the crawler is described in section 9.3.1. Section 9.3.2 describes the manual review method that creates source code patterns from the source code samples.

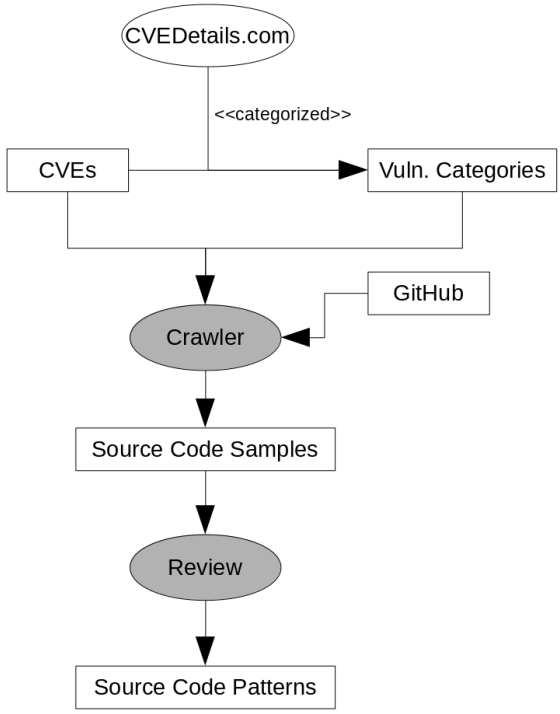


Figure 9.1: Method to determine the source code patterns.

9.3.1 Crawler

A manual review requires a selection of data. We developed a crawler which provides source code samples from open source projects. We focus on vulnerabilities that are tracked in the CVE [82] database. CVE entries between 2010 and 2016 (seven years) are analysed by a crawler. For each CVE, the corresponding entry from *CVE Details* [83] is retrieved. This entry provides categories for each CVE report. 50,765 CVE entries are crawled that have a *CVE details* entry. For each entry, we check if it contains any GitHub [85] *commit* links. This provides the accessibility to the source code. The corresponding *commit* patch also provides code changes which are used to remove the vulnerability. Each *commit* is looked up to determine programming languages based on file extensions of source code files.

9.3.2 Manual review

Each CVE report has a note entry. It usually describes which input fields or variables can be used to exploit the SQL injection vulnerability. Just within the scope these variables/fields are the tainted data sources. The crawler presented in section 9.3.1 searches for CVE report with corresponding GitHub *commit* links. If there is only one link, the GitHub repository revision prior to the *commit* is checked out. This provides the source code for the manual review. The tainted data source is looked up in the source code. Then, the data flow analysis feature of PHPstorm [89] is used to see which parts of the source are reached. If the data flow analysis fails, it will be processed manually. The changes from the *commit* provide a direction whose path is the right one from the source to the sink. Manual review is required to point out the SQL injection vulnerabilities. The review process looks for sources, sanitization methods, string concatenations and sinks [98]. Sources are all kinds of input to the program. For example, this can be a simple PHP parameter. Sanitization methods are all mechanisms that can be used to prevent SQL injections, e.g., by casting a string variable to an integer. String concatenation is investigated to find out what methods are typical and if there are any special cases. Direct database accesses like *execute()* functions are labeled as sinks. Additionally, we explore if the projects have any encapsulation that might mislead developers, for example, developers think these encapsulations already have a protection against SQL injection. So they might think that SQL injections are not possible given a specific encapsulation. Similarities in these patterns are identified and accordingly categories are created accordingly. The categories mentioned in the related work section are factored into the process.

Categories from CVE details	CVEs	Github	PHP	Java	C/C++	js	Python	Ruby
Denial Of Service	10,929	737	7	3	664	4	8	6
Execute Code	10,647	217	70	6	83	8	12	17
Overflow	6,594	302	3	1	280	0	5	1
Obtain Information	5,471	233	32	6	153	5	10	11
Cross Site Scripting	4,878	219	122	5	2	46	14	11
Memory corruption	3,419	52	0	0	49	0	0	0
Bypass a restriction or similar	2,609	96	24	4	51	1	6	2
Gain privileges	2,207	118	4	0	100	0	0	1
Sql Injection	1,732	71	53	3	2	2	0	5
Directory traversal	1,059	35	14	1	7	3	2	5
CSRF	1,046	33	25	2	0	6	1	3
File Inclusion	100	3	0	0	0	0	0	0
Http response splitting	74	4	0	0	0	0	0	0
Summary	50,765	2,120	354	31	1,391	75	58	62

Table 9.1: Software vulnerabilities in open source software grouped by vulnerability type and programming language.

9.4 Results

The results from the manual review show different source code parts for SQL injections in PHP. For sources, sinks, failed sanitization and fixes with successful sanitization the corresponding methods are noted down. For each category, the common methods are clarified in section 9.4.2.

9.4.1 Focus and selection

Figure 9.1 shows the results from the crawler. Only the languages with multiple results are shown. Some CVE entries have multiple categories at once, so these are looked up for each category multiple times. The results show that overflow vulnerabilities are prevalent in C/C++. This is unsuprising because C/C++ are memory unsafe programming languages. [93] The most represented programming language for overflows are C and C++. But C/C++ is missing other important categories like cross site scripting, SQL injection and CSRF. For these categories, PHP is the best language for researching source code patterns. All other languages do not provide enough examples with exception of Javascript for the cross site scripting category. Nevertheless, PHP provides more examples in this category. Accordingly, the selected programming language is PHP. For the research of source code patterns of other languages like Java, more resources are required.

9.4.2 Source code patterns

The reviewing process did provide source code patterns. These patterns were categorized for each interesting source code part. The classification consists of the source, concatenation, sink, sanitization methods from the vulnerable version. Additionally, categories for the fixes from the fixed source code sample are presented.

Categories for sources

For the sources, the following sample categories are identified:

1. **HTTP methods**: *HTTP methods* provide data from a HTTP transfer. In PHP, these methods are found in the standard PHP library. Sources of this kind are commonly in SQL injection examples (e.g., *$_GET[]*, *$_POST[]*, *$_REQUEST[]*).

2. **HTTP wrapper methods**: Some *HTTP methods* have wrappers around them. Some project store the input in collections and the wrapper methods returns corresponding data. Such source code patterns are classified in this category (e.g. *getParam()*).

3. **Environment variables**: Functions in this category provide data from the environment. In this sample set, the function *getenv()* from the standard PHP library was found.

4. **Configurations**: Some functions provide data from configuration files. The function *config_get()* was found. It is a custom implementation to get data out of a configuration file.

5. **Custom functions**: Custom functions are functions that are provided by frameworks or other programming libraries. This category was included because software developers often use third party sources (e.g., callback function parameters).

Categories for concatenations

The classification of the string categorizations contains two categories.

1. **Primitive concatenation**: Source code samples from this category are using the functions provided by features of the programming language. No functions provided by any libraries are used. Examples are the use of variables inside string literals, the use of dot operator and the use of

dot-equals operator (e.g., *"FROM $var"*, *"FROM" . $var*, *$from .= $var*).

2. **Standard functions**: In this sample were the functions *implode()* and *sprintf()* fall into this category. For *sprintf()* function, the type specifier has to be *%s*. Otherwise, the variable will be interpreted as numerical. As seen in the PHP manual, [88] the inserted types would only be double or integer, except for using the *%s* type specifier, which interprets the type as string. Accordingly, the *%s* can lead to a SQL injection.

Categories for sinks

The samples show different sinks. In larger projects, the access to the database is usually encapsulated. Developers are using these encapsulations in the development process. For the sink categories, these encapsulations are treated separately. The following categories are created for the sinks:

1. **Standard database drivers**: This category covers the usage of the standard database drivers without any encapsulation (e.g., *mysqli.query()*, *mysql_fetch_row()*).

2. **Self-implemented database connection**: Such connections cover cases like calling the correct database driver for the utilized database. But connections in this category will not create SQL statements automatically (e.g., *Yii::app->db->createCommand()*, *g_db->Execute()*).

3. **Data-represented objects**: Such objects are software patterns like data access object (DAO) or object-relational mapping (ORM). The tainted data will reach such objects. These objects create SQL statements automatically using stored data (e.g., *ORM->where()*, *$data->setOrder()*).

Categories for fixes

An important part to prevent SQL injections is sanitization. Most reviewed projects had no sanitization before the fix for the CVE entry was developed. Fixes contain different methods to sanitize inputs. The following categories are introduced for the occurring proper sanitization and proper database access:

1. **Standard sanitization methods**: Methods that are provided by the PHP standard library or the database driver libraries fall into this

category (e.g., *addslashes()*, *mysqli_ escape_ string()*).

2. **Custom sanitization methods**: These are methods that are developed by projects. They usually use methods like *preg_ replace()* to replace symbols that are dangerous.

3. **Checks for valid input**: This contains black-listing and white-listing concepts. Projects in this category check if the input contains only valid or any invalid symbols. The input will only reach the sink if the check fits. A common method in this category is *preg_ match()*.

4. **Fixed data types**: For example, casting the input to an integer variable. An integer variable cannot create a SQL injection. The fix for the CVE-2015-4426 used the *sprintf()* function with the *%F* type specifier. This also falls into the category of using fixed data types.

5. **Parametrized code**: Fixes that did use parametrized code for the database access. This is a common construct for creating SQL queries. It separates the data from the query. If this construct is used properly, it prevents any SQL injections (e.g., *mysqli->prepare()*).

Categories for failed sanitization

The last classification contains failed sanitization methods from the vulnerable source code samples. Here, sanitization methods were utilized but they still allowed SQL injection attacks. The following categories were created for failed sanitization methods:

1. **Non-casted variable**: Projects which had this type of sanitization did expect a fixed data type. The variable was casted on the data type and assigned to another variable. The developers did not use casted variable for the SQL query. Instead, they used the variable which was not casted to a fixed data type.

2. **Unexpected control flow**: This category covers sanitization that should change the control flow of the program. However, the method for such a control flow change was misinterpreted by the developers (e.g., the redirect sample from section 9.4.4.).

3. **Comparing different data types**: Some source code samples compared different data types. Based on the data types, different results and interpretations are used (e.g., the comparing variable sample from section 9.4.4).

4. **Incomplete sanitization**: Some tainted data is not sanitized, but the code block uses sanitization methods on other data. For instance, in section 9.4.4 a sample is discussed where this kind of failed sanitization happened because developers did not think more data would be in the array. Also, some samples did check one parameter, but did not check the other parameter for the SQL statement.

5. **No sanitization**: Before the fix was applied, most of the samples had no sanitization methods in the data flow for the vulnerabilities. These samples fall into this category.

9.4.3 Overview

An overview of found source code samples are shown in figure 9.2. Additionally, each classification states the quantity. It is important to note that *CVE-2016-7405* is just a report for an incorrect quote function from the *ADOdb* [81] database class library. Accordingly, this source code sample does not provide a source, concatenation or sink. The *CVE-2011-4960* only states that it allows attacks via unspecified vectors. For this source code sample, the source is ignored. Some CVE reports point out multiple parameters that can be used for an attack. If a report contains patterns that fall into different categories, it is counted in each category.

9.4.4 Special cases

Our data set from the crawler has some source code samples with interesting failed sanitization patterns. Some of these are the only occurrence of the represented category. Nevertheless, they represent mistakes that developers can do in the future. This section points out some of these cases and how they are classified. Figure 9.3 shows an example of a redirect before a SQL statement is executed. This sample falls into the category *unexpected control flow*. Developers probably tought that PHP would be skipped after having set the header location. As the PHP manual [87] gives a hint in a comment to use *exit;* after using the header function, this will prevent that any code afterwards will be run. The function *popupnewsitem()* contains the injectable SQL query. Any attacks are still possible because once the function is called a redirect happens nevertheless. Testing this source code shows that the redirect occurs in browsers. This is a good example for a blind SQL injection. For example, adding a SQL *wait* command will result in a delayed redirect. This can be used to get a response with information based on different delay times.

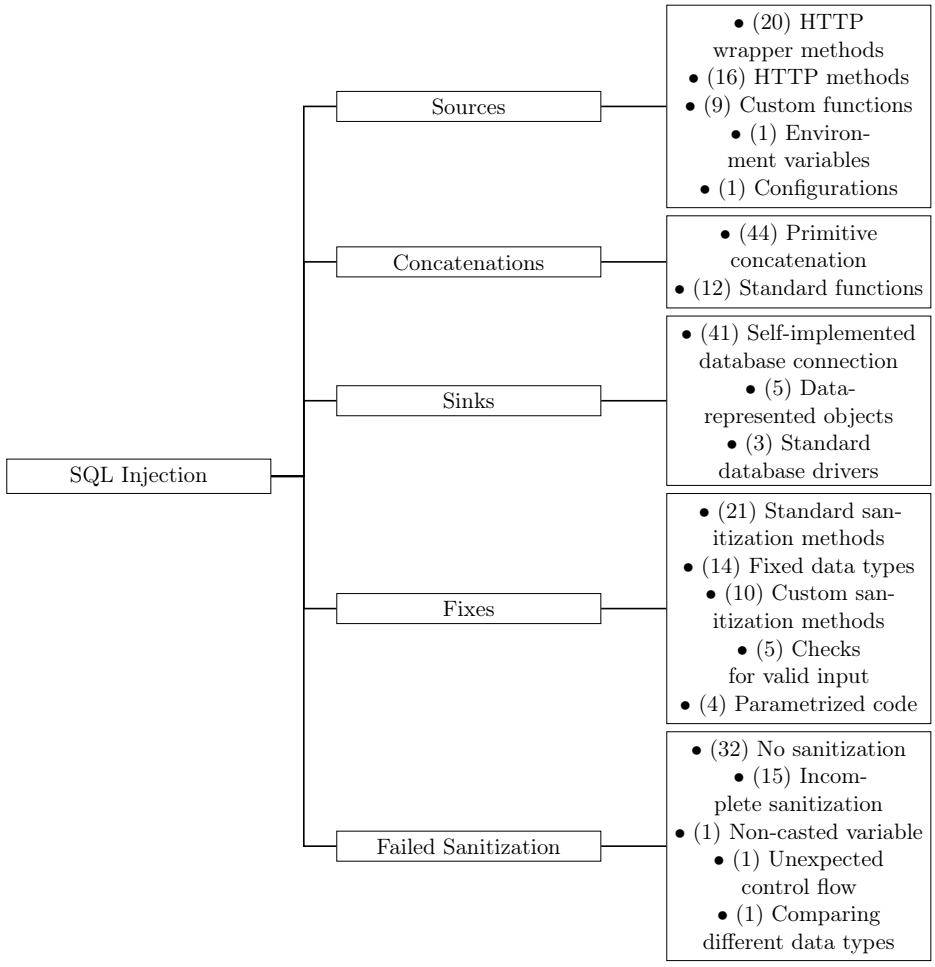


Figure 9.2: Taxonomy of Sql Injection vulnerabilities found in 50 CVE entries related to open source projects.

```
1    public function popupnewsitem($id) {
2
3                if(!is_numeric($id)){header('Location: '.url('/'));}
4
5                $result = PopUpNewsData::popupnewsitem($id);
6                Template::Set('item', $result);
7                Template::Show('popupnews/popupnews_item.tpl');
8        }
```

Figure 9.3: Redirect before SQL statement from CVE-2013-3524.

```
1 function getUserName($userId) {
2   if($userId <= 0) return "Anonymous";
3   $query = "SELECT `user_name` FROM
        `".MYSQL_DATABASE_PREFIX."users` WHERE `user_id` =
        '".$userId."'";
4   $result = mysql_query($query);
5   $row = mysql_fetch_row($result);
6   return $row[0];
7 }
```

Figure 9.4: Comparing variables with different types from CVE-2015-1471.

Another interesting source code sample in figure 9.4 is from the *Comparing different data types* category. The second line shows that the variable *$userId* will be compared with an integer literal. For the comparison, PHP tries to interpret a string variable as a float or integer value. As stated in the PHP manual: "The value is given by the initial portion of the string. If the string starts with valid numeric data, this will be the value used. Otherwise, the value will be 0 (zero)." [90] Accordingly, this sanitization of the user input can be tricked by having numeric data at the beginning of the input. For example, *10' OR 1=1;#* will be interpreted as the number *10*. This example will pass the sanitization and provides the user name of the first user in the database.

The source code from CVE-2014-9089 is shown in figure 9.5. User input will be assigned to an array using the *explode()* function. Line three is the critical one. The developers assumed that the input length will only be two. So they used a fixed size for sanitization. For example, if the exploit uses the third parameter, the sanitization will not be used on that parameter. The fix from the developers used the *array_ slice()* function. This will slice all entries in the array except for the first and the second entry. This ensures that only two sanitized parameters reach the SQL query. Because the inputs values are not checked completely, it falls into the *incomplete sanitization* category.

```
1   $t_sort_fields = explode( ',', $p_filter_arr['sort'] );
2   $t_dir_fields = explode( ',', $p_filter_arr['dir'] );
3   for( $i = 0;$i < 2;$i++ ) {
4      if( isset( $t_sort_fields[$i] ) ) {
5         $t_drop = false;
6         $t_sort = $t_sort_fields[$i];
7         if( strpos( $t_sort, 'custom_' ) === 0 ) {
8            if( false === custom_field_get_id_from_name(
                 utf8_substr( $t_sort, utf8_strlen( 'custom_' ) ) ) )
                 {
9               $t_drop = true;
10           }
11        } else {
12           if( !in_array( $t_sort, $t_fields ) ) {
13              $t_drop = true;
14           }
15        }
16        if( !in_array( $t_dir_fields[$i], array( "ASC", "DESC" ) )
              ) {
17           $t_drop = true;
18        }
19        if( $t_drop ) {
20           unset( $t_sort_fields[$i] );
21           unset( $t_dir_fields[$i] );
22        }
23     }
24  }
```

Figure 9.5: Sanitization on fixed numbers of inputs from CVE-2014-9089.

9.5 Discussion

The results show different kinds of source code patterns that result in a SQL injection vulnerability. The resulting categories allow to create different source code permutations of SQL injections. Many source code parts from relating CVE reports did not have any sanitization of the inputs. Many developers probably require a better software security education to prevent such issues. For example, the source code from CVE-2014-8351 looked like a classical teaching example of a SQL vulnerability. All relevant parts of the SQL injection (source, concatenation and sink) were not even in ten lines of code. This shows that software security education is indispensable.

The special case with the redirect from figure 9.3 showed that knowledge about the used API is very important. Developers have to know how the API and programming language work. But in this special case, the API description of the PHP manual [87] is not perfect either. The usage of the *exit()* function was mentioned in the code example as a comment only. Maybe this should be mentioned more clearly such that it keeps developers from creating security issues or bugs in this context. At least in our reviewed code sample, the developers did think about security and checked the input.

The work done in [101, 102] describes categories for sources, sanitization and sinks. For the sources, they have *Client* which is the same category as *HTTP methods*. They also mention sources from databases. Such source code patterns were not found in our data set. However, this is a more common source for cross site scripting than for SQL injection. They also have *Persistent* and *Uninit* as data sources which were not found in our data set. Our new categories for sources are *Environment variables*, *Configurations* and *Custom functions*. The focus of their work is on the sanitization methods. They did categorize these methods more deeply into the functionality. For example, *LimitLength* are functions that limit the length of the input. *Encryption* are a category where methods like *sha1()* are used. No source code patterns with such sanitization methods are found in the data set. More subcategories for *Checks for valid input* are found in Medeiros et al. [99]. They presented categories for black-listing, white-listing, replace string, et cetera. Our results show that these kinds of sanitization methods are used in open source projects. We added the categories for failed sanitization methods which are a new perspective on the use of sanitization methods.

9.6 Conclusions and future work

To minimize the occurrence of SQL injections, different source code patterns have to be detected and avoided. We analysed the source code of 50 GitHub projects which are correlated to CVE reports that mention SQL injection and provide available source code fixes. The focus was on how the vulnerabilities appeared in real projects. For each important part of the pattern, categories were created based on the analysed source code. Compared to related work, our results show what kind of software vulnerabilities appeared in open source projects. We identified new categories not present in earlier work and found existing categories to be less frequently used than expected. The knowledge of source code patterns allows to create exercises with the same patterns that can be used to teach developers software security skills. An interesting point will be to create these exercises automatically. Existing source code from projects can be used and transformed to create these source code patterns. This will be an important research because malicious developers might program such tools. This would reveal some limitations and maybe risks that might occur by such tools in the future. Another part will be using these categories to test static code analysis tools. Test static analysis tools and investigate whether they detect all permutations.

References

[81] ADOdb, 2017. URL `https://github.com/ADOdb/ADOdb`.

[82] Common Vulnerabilities and Exposures, 2017. URL `https://cve.mitre.org/`.

[83] CVE Details, 2017. URL `https://www.cvedetails.com/`.

[84] Common Weakness Enumeration, 2017. URL `https://cwe.mitre.org/`.

[85] GitHub, 2017. URL `https://github.com/`.

[86] OWASP Top Ten Project, 2017. URL `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[87] PHP manual - function.header.php, 2017. URL `http://php.net/manual/en/function.header.php`.

[88] PHP manual - sprintf, 2017. URL `http://php.net/manual/en/function.sprintf.php`.

[89] PhpStorm, 2017. URL https://www.jetbrains.com/phpstorm.

[90] PHP manual - string conversion, 2017. URL http://php.net/manual/en/language.types.string.php\#language.types.string.conversion.

[91] SAMATE - SARD, 2017. URL https://samate.nist.gov/SARD.

[92] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb. Security analysis and enhancements of computer operating systems. Technical report, DTIC Document, 1976.

[93] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. *SIGPLAN Not.*, 41(6):158–168, June 2006.

[94] R. Bisbey and D. Hollingworth. Protection analysis: Final report. *University of Southern California Information*, 90291(2223), 1978.

[95] Z. Hui, S. Huang, B. Hu, and Z. Ren. A taxonomy of software security defects for SST. *Proceedings - 2010 International Conference on Intelligent Computing and Integrated Systems, ICISS2010*, pages 99–103, 2010.

[96] P. Lerthathairat and N. Prompoon. An approach for source code classification to enhance maintainability. *Proceedings of the 2011 8th International Joint Conference on Computer Science and Software Engineering, JCSSE 2011*, pages 319–324, 2011.

[97] F. Massacci and V. H. Nguyen. Which is the right source for vulnerability studies?: An empirical analysis on Mozilla Firefox. *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, pages 4:1–4:8, 2010.

[98] G. McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[99] I. Medeiros, N. Neves, and M. Correia. Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining. *IEEE Transactions on Reliability*, 65(1):54–69, 2016.

[100] R. C. Seacord and A. D. Householder. A Structured Approach to Classifying Security Vulnerabilities. *Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst*, (January):1–39, 2005.

[101] L. K. Shar and H. K. Tan. Mining Input Sanitization Patterns for Predicting SQL Injection and Cross Site Scripting Vulnerabilities. pages 1293–1296, 2012.

[102] L. K. Shar, H. Beng Kuan Tan, and L. C. Briand. Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis. *Proceedings - International Conference on Software Engineering*, pages 642–651, 2013.

[103] B. Stivalet and E. Fong. Large Scale Generation of Complex and Faulty PHP Test Cases. *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, pages 409–415, 2016.

[104] Y. Wu, H. Siy, and R. Gandhi. Empirical results on the study of software vulnerabilities. *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 964–967, 2011.

Chapter 10

Source Code Patterns of Buffer Overflow Vulnerabilities in Firefox

10.1 Introduction

The Common Weakness Enumeration (CWE) [109] top 25 show buffer overflow vulnerabilities (CWE-120) in third place. Buffer overflows have existed for a long time. To discover the reason why buffer overflows still occur in code, we investigated source code samples from the open source web browser Firefox [111]. Different categories for buffer overflow vulnerabilities already exist in CWE. These categories take a technical point of view; they look at aspects such as which memory locations are involved. For example, there are categories for accessing memory on the stack or on the heap. Such categories do not help software developers to avoid buffer overflow vulnerabilities. Developers have to know how vulnerabilities occur and what kind of source code patterns are common for vulnerabilities. Additionally, developers have to know how vulnerabilities can be mitigated. For example, it is important to check inputs carefully and to not misuse memory-critical functions as *memcpy()* that is listed as on of security development lifecycle banned function calls from Microsoft [116]. To fill the gap in the current categorization approaches and provide a structured body of knowledge for software developers to mitigate buffer overflow vulnerabilities, we reviewed 50 source code samples of buffer overflow vulnerabilities in Firefox. In our review, we considered which types of errors the developers made, which sinks were involved in buffer overflow vulnerabilities and how developers patched

the vulnerability. Categories were created based on the results from the reviews. These results are compared to the categories from CWE to see the difference from a developer's point of view.

This paper begins with an overview of related work in section 10.2. The following section explains how the source code was obtained as well as the review method. The categories for buffer overflows are presented in sections 10.4, 10.5 and 10.6. The last two sections provide a discussion of the results and suggestions for future work.

10.2 Previous and related work

SQL injection vulnerabilities in 50 source code samples from open source projects were analysed by [119] using a similar method. The reviewed programming language was PHP. Classifications of source code patterns exist. Classifying source code into bad code, clean code and ambiguous code was done by Lerthathairat and Prompoon [114]. Metrics in source code like comments, the size of the function, et cetera. were analysed using fuzzy logic to determine which category the source code belongs to. Bad and ambiguous code are considered for refactoring. More security-related work is by Hui et al. [112], using a software security taxonomy for software security tests. The security defects taxonomy was created based on the top 10 software security errors from authoritative vulnerability enumerations. It is categorized into into *induced causes*, *modification methods* and *reverse use methods*. Hui et al. [112] suggested to use their taxonomy as security test cases.

Massacci and Nguyen [115] investigated different data sources for vulnerabilities, e.g. Common Vulnerabilities and Exposures (CVE) [108], National Vulnerability Database (NVD) [117], et cetera. They checked which data sources were used by other research projects. In their work, Firefox was used as database for the analysis. Semantic templates were derived from the existing CWE database and are supposed to help understand security vulnerabilities by Wu et al. [122]. The authors did an empirical study to prove that these semantic templates have a positive impact on the time until a vulnerability is completely found.

The work by Bishop et al. [106] [105] presents a taxonomy of how buffer overflow vulnerabilities occur, considering which preconditions are required to exploit a vulnerability. These preconditions are not suited to teach software developers to mitigate vulnerabilities. For example, taking into consideration the category that the program can jump to a memory location in the stack. This is relevant for exploiting the vulnerability, but it will not help to understand what kind of mistakes were done in developing the source

code. Kratkiewicz and Lippmann [113] used a taxonomy of buffer overflow vulnerabilities to create a data set of 291 small C programs. The data set was analysed with static and dynamic code analysis tools. The tools were then evaluated regarding their detection rate, false positive rate, et ceterea. Ye et al. [123] analysed 100 buffer overflow vulnerabilities and the corresponding patches, using the data to evaluate static code analysis tools. The evaluated tools were *Fortify*, *Checkmarx* and *Splint*. Shahriar and Haddad [120] showed how to automatically patch buffer overflow vulnerabilities, including a classification of different types of buffer overflow vulnerabilities. For each of these categories, rules were offered to mitigate the vulnerability. The SEI CERT coding standards [121] provide an overview on how to implement memory-critical parts in C. The standards are presented as necessary to ensure safety, reliability and security. Non-compliant and compliant code examples help developers to better understand the coding standards.

10.3 Method

To create the source code pattern categories, selected data sets are needed for the review process. We focus on vulnerabilities which are tracked in the CVE database. We chose source code samples from Firefox because it has many reported buffer overflow vulnerabilities - on average about 30 buffer overflow vulnerabilities per year. Additionally, the Bugzilla [107] platform offers a public discussion about the bug fixes, which helps to identify the relevant source code pattern for the vulnerability. 187 CVE reports are connected to buffer overflow vulnerabilities and Firefox in the time frame from 2010 to 2015 (six years). We choose 2015 as a cut-off to ensure we would have access to a patch as well. We use 50 randomly selected CVE reports which also provide a patch to fix the vulnerability. The patch is determined by a *CONFIRM* flag in the CVE report which indicates the correct Bugzilla report. The bug report contains a link to the patch which fixes the vulnerability. Firefox patches are managed with a version control tool. For each of theses patches, the hash value of the parent version is specified. That version is used as a source code sample containing the buffer overflow vulnerability.

The vulnerable version was reviewed regarding the types of errors made by developers and which sinks were used. A sink is the last instance where unchecked user input can exploit a vulnerability. For example, the function *memcpy()* is a common sink for buffer overflow vulnerabilities. In order to see which errors were made and which sinks were used, a data flow analysis was performed. This was done manually because within the bounds of our project, we could not find a proper tool that was able to analyse such a huge

project like Firefox. It is possible that types of errors appear several times because a combinations of errors can also result in a vulnerability. The errors were considered from a developer's point of view. However, the sink is more focused on which critical functions and source code parts are used. This helps developers to recognize critical functions. The patch was used to see how developers fixed the vulnerability. The review of the patch was used to create categories for the fixes.

10.4 Types of errors

Figure 10.1 shows an overview of the categories for the types of errors found in our data set. It has more than 50 assignments because one type of error can lead to other types of errors which then result in a buffer overflow vulnerability. The categories created for the data are the following:

1. ***Variable Overflow***: Many instances of buffer overflows in our review are correlated to integer overflows or underflows. These over-/underflowed variables are used to check the input size (*Variable Overflow in Check*). Because of the wrong value, these checks pass inputs resulting in a buffer overflow condition. This type of error can be represented in a combination of the following CWE ids: The CWE-190 (Integer Overflow or Wraparound) connected with the keyword *CanProcede* to CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer).

   An overflowed or underflowed variable is used for allocating memory (*Variable Overflow Allocation*). The allocated memory is smaller than the input copied into it. This results in a buffer overflow. The related CWE id is CWE-680, which states that a calculation result is used to allocate memory and an integer overflow causes less memory to be allocated. The allocation of an insufficient amount of memory in our data set occurred in the following sub-patterns:

   (a) *Allocation too small*: An integer overflow can either have a negative result (signed int) or very small result (unsigned int). These integer overflows occur because user data is included in a calculation. This can be a simple addition to a static value or it can be methods computing a length. As an example, the length of the user input could be the sum of multiple user inputs. If memory is allocated from an integer overflow result, the later usage of the memory will result in a buffer overflow vulnerability.

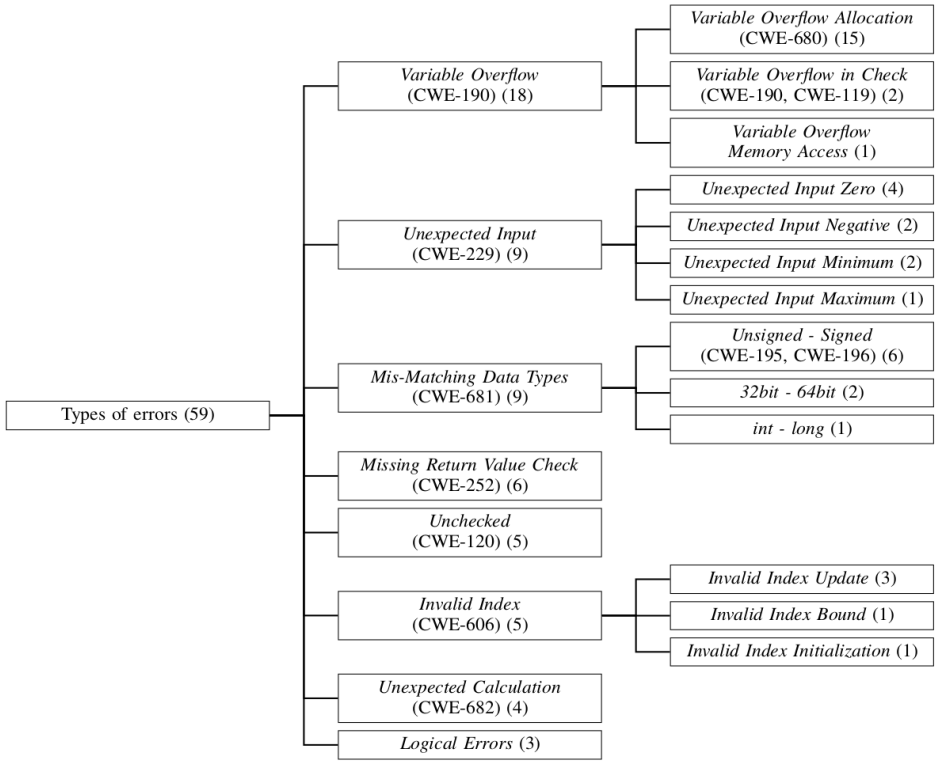


Figure 10.1: Taxonomy of errors developers introduced based on the data set.

(b) *Existing buffer size check*: Some data sets used already existing buffers and checked if the buffer size had to be increased. An integer overflow in such a check also results in a buffer that is too small.

2. ***Unchecked***: The review shows vulnerabilities where user input reaches methods that are vulnerable for buffer overflows. Source code samples without any checks fall into this category. The corresponding CWE id (CWE-120) explains it as follows: "'The program copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow."' Accordingly, this is the classic buffer overflow where input is not checked and then is able to reach critical functions like *memcpy()*.

3. ***Unexpected Input***: This category covers unexpected user inputs. Usually, all of the error types could fall into this category, but it covers inputs that the developers did not expect to occur. For example, the parameter of a method is the size of a file. Accordingly, the parameter must not be negative (*Unexpected Input Negative*). Another example would be a parameter that has a minimum size, thus, falls into the category *Unexpected Input Minimum*. Nevertheless, the parameter can be outside of the expected range because of some other preconditions or special inputs. For example, a specially crafted file would return a negative result as the content length. If a developer uses such premises for memory-critical parts, a buffer overflow vulnerability could occur. One sample also had expected a maximum input (*Unexpected Input Maximum*) of a value. This vulnerability was related to shaders programs which are programs running on the graphics processor. Developers did not think that the value of the input could be higher than the number of existing shaders. CWE-229 (Improper Handling of Values) is best suited to our *Unexpected Input* category because the inputs are not handled properly. The CWE category covers multiple variants like missing values or undefined values. It does not cover numerical values which are too small, too high or in an unexpected range.

4. ***Mis-Matching Data Types***: This category covers vulnerabilities where values of different data types are assigned to each other which is presented in CWE-681 (Incorrect Conversion between Numeric Types). A common example is the assignment from *unsigned int* to *signed int*. These assignments are also covered by CWE-119 (Signed to Unsigned Conversion Error) and CWE-196 (Unsigned to Signed Conversion

Error). This type of error occurred in our data set in combination with *Unexpected Calculation* or just as a simple conversion with the outcome of a buffer overflow vulnerability. Also, some samples contain assignments of different variable lengths, for example, assignments between 32 bit and 64 bit variables. C/C++ does allow the assignment of variables with different data types. It will interpret the bits according to the new variable type. For example, if a negative value is assigned to an unsigned variable, the first bit will be interpreted as the highest value bit. If such an interpretation is unwanted, subsequent checks and the usage of the variable will be problematic. In our sample, this results in buffer overflow vulnerabilities.

5. ***Missing Return Value Check***: These are vulnerabilities where developers do not check the return value. In our data set it was common that the return value of a memory allocation function was not checked. If the allocation is not possible, the allocation functions returns an error code. If the return value is ignored, the pointer will point to a random memory address. Using this pointer to access memory will likely result in a buffer overflow vulnerability. Usually such a situation only happens when the system or program is out of memory. CWE-252 (Unchecked Return Value) is the related category in the CWE list.

6. ***Invalid Index***: These error types include the usage of an invalid index for a loop. It is split into three subcategories. The first is the *Invalid Index Bound* where the bound is invalid. This can happen because of previous errors like an *Unexpected Calculation*. Samples where such a bound is invalid and the index is used to access memory are counted in this category. Another error is that developers did not update the index correctly (*Invalid Index Update*) which also results in a buffer overflow. One sample had an index initialized to an invalid value (*Invalid Index Initialization*). The best fitting CWE category is CWE-606 (Unchecked Input for Loop Condition) because the *Invalid Index* category is related to loops.

7. ***Unexpected Calculation***: This category covers source code samples where unexpected results are obtained during calculation. All the samples had a negative result. The developers did not expect the result to be negative and the values were used in memory-unsafe functions. Another example is assigning a negative result to an unsigned integer. The unsigned integer will interpret the highest bit which is a 1 as

a very large value because it was negative when it was represented in a signed datatype. Such an example is represented in CWE ids with the following: CWE-682 (Incorrect Calculation) connected with the keyword *CanFollow* to CWE-681 (Incorrect Conversion between Numeric Types).

8. ***Logical Errors***: Two vulnerable samples showed developers made logical errors. For example, not enough memory was allocated regardless of the input and the following code did write into unintended memory parts. Another sample had an issue where the length of a variable was not updated correctly and that length was used in memory-critical parts. Three samples showed buffer overflow conditions because they had logical errors.

10.5 Sink categories

What kind of sinks were used in the data set are shown in figure 10.2. These are classified into the following categories:

1. ***Critical Functions***: Sinks of this category are memory-critical functions. Common functions in C/C++ are *memcpy()* or *memset()*. These functions are categorized into the subcategory *transfer memory*. Three sinks of the data set used a string copy function (*strcpy()*) and one sample used the *scanf()* function. These are functions which are also found in the banned functions list for security development lifecycle [116].

2. ***Array***: Arrays in C/C++ are very similar to pointers. The memory for an array is arranged such that all entries are next to each other. If an array field is accessed using an invalid index, a buffer overflow vulnerability exists. All data sets where the sinks are arrays fall into this category. This can be split into write (CWE-787: Out-of-bounds write) and read (CWE-125: Out-of-bounds read). Two samples performed a read access with a static index. Both of them used the index zero, which is typically used to get the first element of an array.

3. ***Pointer***: The last category for sinks is the misuse of pointers. These are sinks where pointers are used to access memory. This category can be mapped to the CWE-468 (Incorrect Pointer Scaling) category. This category can also be split into subcategories of read (CWE-125: Out-of-bounds read) and write (CWE-787: Out-of-bounds write).

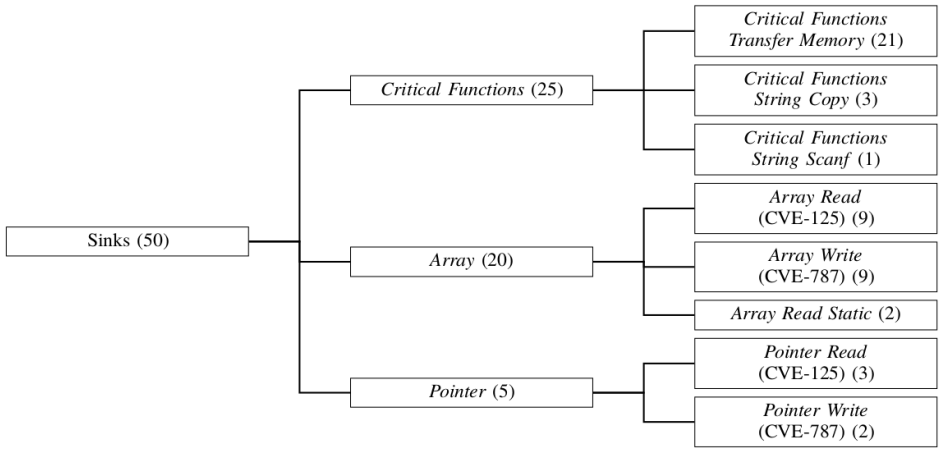


Figure 10.2: Taxonomy of sinks based on the data set.

10.6 Fix categories

The results for the different fixes are categorized and seen in figure 10.3. They are connected to the different problem types. Fixes are categorized as follows:

1. ***Proper Input Check***: Fixes for this category are input checks which were completely missing (*Unchecked*). Also fixes which check inputs that developers didn't have in mind fall into this category (*Unexpected Input*). The subcategories are for the different kinds of checks. For example, negative values are a common input developers did not expect. Also some vulnerabilities which have *Variable Overflow in Check* and *Variable Overflow Allocation* as error categories were fixed by checking if the input value was not too high. Also some fixes did just check if a value was not too small. This is commonly a fix when developers thought the input could not be that small. Black listing where specific inputs are filtered out and white listing where only specific inputs are allowed were found as fixes in the data set.

2. ***Check Overflow Underflow***: Firefox has a *checkedint* class which allows to check if an overflow or underflow occurred. Accordingly, fixes used these classes instead of *int* variables and checked for over- and underflow occurrences. Two fixes did check the input, i.e., if an integer overflow occurred in the calculation before using it. This fixes problems from the *Variable Overflow in Check* and *Variable Overflow Allocation*

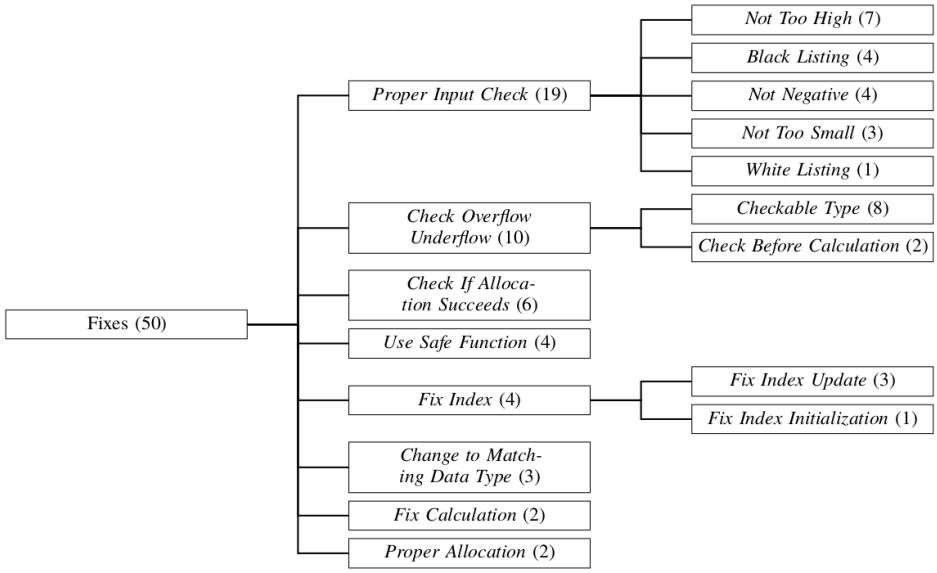


Figure 10.3: Taxonomy of fixes for the vulnerabilities based on the data set.

categories.

3. ***Check If Allocation Succeeds***: Some vulnerabilities fall in the error category *Missing Return Value Check*. In our data set, these missing return value checks are related to allocating memory. As the name already hints, fixes in this category check these return values and change the control path accordingly.

4. ***Use Safe Function***: Memory related functions provide a secure function which requires an additional parameter. This parameter is used to restrict the size which is used in the memory-critical function. A common example is *strcpy* and *strcpy_s*. The additional parameter is used to provide the size of the string. This prevents a vulnerability where the source string has no null character or the size of the source string is larger than the size of the destination string. Four fixes used such functions to remove the vulnerability.

5. ***Fix Index***: This category is related to the error type *Invalid Index*. These errors were fixed by using valid indexes. One instance was fixed by changing the data type so that the index was not invalid any more. The fixes are split into the same subcategories as the error type. For example, one index update fix was implemented by inserting a *break*

statement.

6. ***Change to Matching Data Type***: Three vulnerabilities were patched by changing the data type. This fix is related to the *Mis-Matching Data Types* error type category. Three of these kind of errors were patched by changing the data type. The remaining samples were fixed by correcting a previous error which then only resulted in a vulnerability because of mis-matching data types. For example, an integer overflow was fixed, which resulted in a negative result that was assigned to an unsigned integer variable. As long as the value was positive, this did not create a problem.

7. ***Fix Calculation*** Two samples were patched by fixing the calculation. The calculation was adjusted accordingly such that the undesired results will no longer occur.

8. ***Proper Allocation*** The last category of fixes are patches where the allocations were fixed. For example, the allocation did not reserve enough memory. If the allocation was changed such that it allocated the right amount of memory, it falls into this category. Two samples patched the vulnerability by correctly allocating memory.

10.7 Discussion

Firefox is a well-known open source product and the source code is reviewed a lot. Accordingly, the vulnerabilities from Firefox usually had input checks before potentially dangerous functions or memory accesses were used. The vulnerabilities most often existed because an integer overflow or underflow occurred. It is important to teach developers the right use of variables which may overflow/underflow. Also the assignment of variables with different data types in *C/C++* is problematic and should be avoided. Nevertheless, if these assignments are required, they should be used carefully.

The error types were related to existing CWE categories. CWE-888 contains software fault pattern clusters. The containing category CWE-890 (SFP Primary Cluster: Memory Access) is related to buffer overflow vulnerabilities. This category also has the following subcategories:

- CWE-970 SFP 2. Cluster: Faulty Buffer Access: **covered**
- CWE-971 SFP 2. Cluster: Faulty Pointer Use: **did not occur in data set**

- CWE-972 SFP 2. Cluster: Faulty String Expansion: **did not occur in data set**
- CWE-973 SFP 2. Cluster: Improper NULL Termination: **did not occur in data set**
- CWE-974 SFP 2. Cluster: Incorrect Buffer Length Computation: **covered**

CWE-970 and CWE-974 are covered by our data set. Surprisingly, CWE-971 did not occur in our data set. This is due to the fact that this category has only very specific CWE subcategories, for example, when using a *null* pointer or using pointers to determine a length. Also CWE-972 and CWE-973 did not occur in our data set. There was no vulnerability sample related to an improper *null* termination of a *String* variable. Another related cluster is CWE-969 (SFP Secondary Cluster: Faulty Memory Release) which covers vulnerabilities where memory is released and still used later on. This includes vulnerabilities like "'double free"' or releasing memory which is not on the heap. Unfortunately, our data set did not cover vulnerabilities which fit into this cluster.

As already stated, Microsoft released a list of banned functions for the security development lifecycle [116]. Most of our sinks that fall into the category *Critical Functions* are found on the list. Our data set contained the critical functions *memmove()* and *memset()*, which are not found in the banned list because these functions are using a restricting length parameter. Only four of samples using a banned function were fixed by using a safe function. 17 of the sinks in our data set did use the function *memcpy()*. According to the list, the function *memcpy_s()* should be used which requires an additional parameter defining the size of destination. None of the patches used the function to fix the vulnerability. It is easy to tell developers to avoid buffer overflow vulnerabilities, but there is a huge list of critical functions. Developers have to know which functions are critical. Static code analysis tools might be useful to find these functions. Nevertheless, in our data set only half of the vulnerabilities use critical functions. Additionally, there are many different permutations of buffer overflow vulnerabilities which makes the mitigation for developers problematic.

Our results show that buffer overflow vulnerabilities are not simply avoided by having a list of critical functions. Buffer overflow vulnerabilities occur in many different permutations and in combination of errors. Accordingly they are not easy to prevent by just learning simple vulnerabilities. Our results provide an overview of source code patterns which are found in our data set. These can be used to teach developers that all kinds of permutations of our

categories can result in a buffer overflow vulnerabilities.

10.8 Conclusions and future work

To minimize the occurrence of buffer overflow vulnerabilities, different source code patterns have to be detected and avoided. To gain a better understanding of how such patterns look like, we analysed 50 buffer overflow CVE reports related to Firefox. We created categories for the types of errors the developers made, what kind of sinks were used and how the developers fixed the vulnerability. These categories were compared to existing CWE categories. Some categories are not found as a direct CWE category. Likewise, our data set does not include all CWE categories. The focus of the categories is seen from a developer's point of view instead of a technical representation of the vulnerability details. This helps to use the categories to teach developers which source code patterns and errors are common for buffer overflow vulnerabilities.

Our patterns could be used to create different learning exercises using different permutations. An interesting point will be to create these exercises automatically. The LAVA tool [110] injects buffer overflow vulnerabilities in C code. It would be interesting to integrate our patterns into this tool. This will be an important step because malicious developers might already have developed such tools. It would reveal some limitations and maybe risks which might occur by automatically creating vulnerabilities in the future. Our earlier work [118] is a tool which injects SQL injection vulnerabilities in Java source code using an abstract syntax tree. A similar approach might be possible to inject buffer overflow vulnerabilities in C/C++ code. Another avenue of research would be using these categories to benchmark static code analysis tools. Data sets could be created using different permutations of our categories. It will be interesting to see if all permutations are detected by static code analysis tools as well as the false positives and the false negatives rates of the tools.

References

[105] M. Bishop, D. Howard, S. Engle, and S. Whalen. A taxonomy of buffer overflow preconditions. 2010.

[106] M. Bishop, S. Engle, D. Howard, and S. Whalen. A taxonomy of buffer overflow characteristics. volume 9, pages 305–317, 2012.

[107] Bugzilla, 2017. URL `https://bugzilla.mozilla.org`.

[108] Common Vulnerabilities and Exposures, 2017. URL https://cve.mitre.org/.

[109] Common Weakness Enumeration, 2017. URL https://cwe.mitre.org/.

[110] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 110–121. IEEE, 2016.

[111] Firefox, 2017. URL https://www.mozilla.org/de/firefox/.

[112] Z. Hui, S. Huang, B. Hu, and Z. Ren. A taxonomy of software security defects for SST. pages 99–103, 2010.

[113] K. Kratkiewicz and R. Lippmann. A taxonomy of buffer overflows for evaluating static and dynamic software testing tools. In *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*, pages 500–265, 2005.

[114] P. Lerthathairat and N. Prompoon. An approach for source code classification to enhance maintainability. pages 319–324, 2011.

[115] F. Massacci and V. H. Nguyen. Which is the right source for vulnerability studies?: An empirical analysis on Mozilla Firefox. pages 4:1—-4:8, 2010. ISBN 978-1-4503-0340-8.

[116] Microsoft: Security Development Lifecycle (SDL) Banned Function Calls, 2017. URL https://msdn.microsoft.com/en-us/library/bb288454.aspx.

[117] National Vulnerability Database, 2017. URL https://nvd.nist.gov/.

[118] F. Schuckert. PT: Generating Security Vulnerabilities in Source Code. In *Sicherheit 2016 - Sicherheit, Schutz und Zuverlässigkeit*, pages 177–182, 2016.

[119] F. Schuckert, B. Katt, and H. Langweg. Source code patterns of sql injection vulnerabilities. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, pages 72:1–72:7. ACM, 2017.

[120] H. Shahriar and H. M. Haddad. Rule-based source level patching of buffer overflow vulnerabilities. pages 627–632, 2013.

[121] C. C. Standard. Sei cert. 2016.

[122] Y. Wu, H. Siy, and R. Gandhi. Empirical results on the study of software vulnerabilities. pages 964–967, 2011.

[123] T. Ye, L. Zhang, L. Wang, and X. Li. An Empirical Study on Detecting and Fixing Buffer Overflow Bugs. pages 91–101, 2016.

Chapter 11

Source Code Patterns of Cross Site Scripting in PHP Open Source Projects

11.1 Introduction

Cross Site Scripting (XSS) is on the fourth place in Common Weakness Enumeration (CWE) top 25 2011 [126] and on the seventh place in Open Wep Application Security Project (OWASP) top 10 2017 [127]. Accordingly, Cross Site Scripting is still a common issue in web security. To discover the reason why the same vulnerabilities are still occurring, we investigated the vulnerable and patched source code from open source projects. Similar methods, functions and operations are grouped together and are called source code patterns. Our work shows which source code patterns occur in real life projects, to provide a data set that can be compared to existing vulnerability data sets like SAMATE SARD [131]. The questions to be answered are: How do source code patterns from real projects look like? What main protection mechanisms are required to prevent Cross Site Scripting attacks? Real source code samples from open source projects are investigated to get answers to these questions.

This paper begins with an overview of related work in section 11.2. Section 11.3 explains how we got the source code sample and how the manual review process looks like. The next section explains what Cross Site Scripting categories from CWE [126] exist and how they fit into source code pattern categories. In section 11.5 our taxonomy resulting from the manual review process is explained. The sample from CVE-2012-5163 described in section

11.6 is a sample where developers might thought that data is already sanitized. In section 11.7 the results are discussed. The final section provides some discussion about how our result can be used in the future.

11.2 Related Work

For SQL injection and Buffer Overflow vulnerabilities, we created the source code patterns with the same method that was used by [146] and [145]. Research projects about general classification of source code patterns exist. Lerthathairat and Prompoon [138] did research about the classification of source code into bad code, clean code and ambiguous code. They use metrics in source code like comments, the size of the function, et cetera. These metrics were analysed by using Fuzzy logic to determine which category the source code belongs to. Bad and ambiguous code were improved by refactoring. A more security related work is from Hui et al. [137], they use a software security taxonomy for software security tests. They created a security defects taxonomy based on top 10 software security errors from authoritative vulnerabilities enumerations. Their taxonomy is categorized into *induced causes*, *modification methods* and *reverse use methods*. They advise that their taxonomy should be used as security test cases. Stivalet and Fong [147] present a tool that allows to create short code examples including software vulnerabilities. They split up the source code parts into *input*, *filtering* and *sink*. Permutations of these parts will create different examples. All of the examples can be found in the Testsuite 103 within the Software Assurance Reference Dataset [131].

The research also requires data sources, which are used to classify the source code patterns. Massacci and Nguyen [140] researched different data sources for vulnerabilities, e.g. Common Vulnerabilities and Exposures (CVE), National Vulnerability Database (NVD), et cetera. They looked into which data sources were used by other research projects. They also used Firefox as a database for their analysis. Wu et al. [148] use semantic templates created from the existing CWE database [126]. They should help to understand security vulnerabilities. The authors did an empirical study to prove that these semantic templates have a positive impact on the time to completion on finding the vulnerability.

Louw and Venkatakrishnan [139] present a tool Blueprint can be used to defend against Cross Site Scripting attacks. Different Cross Site Scripting variants are explained and how they can be exploited. Another framework from Gupta and Gupta [136] can be used to protect against XSS attacks specialized on HTML 5 web pages. Another approach from Maurya [141]

[142] shows how to prevent Cross Site Scripting vulnerabilities on the server side. In their work different security levels require different sanitization approaches. Nadji et al. [143] present different XSS attack scenarios and also suggest on how to prevent these attacks using a combination of server and client protection mechanisms. Another work from Gundy et al. [135] uses a randomized approach to prevent against XSS attacks. As long as the attackers cannot predict the randomization that approach should deny any XSS exploits. In contrast we suggest using the correct standard prevention methods based on the html context to prevent XSS attacks.

11.3 Method

For our research we decided to review vulnerabilities that were found in open source projects. We focus on vulnerabilities that are tracked in the *CVE* [124] database. CVE reports between 2010 and 2016 (seven years) are used as data samples to ensure that developers had sufficient time to patch the vulnerabilities. To get the vulnerable and patched source code related to CVE reports the results from the source code crawler from [145] were used. It checks CVE entries for related GitHub [125] patches and downloads the vulnerable and patched source code. Table 11.1 shows how many source code samples for different vulnerability types and programming languages are found. We chose PHP as reviewed programming language because it provides the most samples (122) related to Cross Site Scripting. Out of these source code samples 50 CVE reports are randomly selected for a manual review. 50 samples means 1% of all CVE entries related to XSS and 40% of all CVE entries related to XSS and PHP.

Cross Site Scripting vulnerabilities are commonly split into three parts (sources, sanitization and sinks). The sources are methods where data is provided which can be manipulated by a user. Sinks are methods where such data can be harmful. In a XSS vulnerability it will result in scripts that will be executed on the victim's browser. Sanitization methods transform user provided data such that it will not be harmful, if it reaches sinks.

The manual reviews were done as follows. For each CVE report the note entry was looked up. It usually describes which input fields or variables can be used to exploit the vulnerability. Just within the scope these variables/fields are the tainted data sources for the Cross Site Scripting vulnerabilities. If no variable/field is mentioned the source code is manually backtracked from the patched source code to find the sources. By doing a data flow analysis from the sources relevant source code patterns are tracked. For example, it will be noted if a sanitization method from a framework is used. For sources,

Categories from CVE details	CVEs	Github	PHP	Java	C/C++	js	Python	Ruby
Denial Of Service	10,929	737	7	3	664	4	8	6
Execute Code	10,647	217	70	6	83	8	12	17
Overflow	6,594	302	3	1	280	0	5	1
Obtain Information	5,471	233	32	6	153	5	10	11
Cross Site Scripting	4,878	219	122	5	2	46	14	11
Memory corruption	3,419	52	0	0	49	0	0	0
Bypass a restriction or similar	2,609	96	24	4	51	1	6	2
Gain privileges	2,207	118	4	0	100	0	0	1
SQL Injection	1,732	71	53	3	2	2	0	5
Directory traversal	1,059	35	14	1	7	3	2	5
CSRF	1,046	33	25	2	0	6	1	3
File Inclusion	100	3	0	0	0	0	0	0
Http response splitting	74	4	0	0	0	0	0	0
Summary	50,765	2,120	354	31	1,391	75	58	62

Table 11.1: Software vulnerabilities in open source software grouped by vulnerability type and programming language.

insufficient sanitization, PHP sinks, HTML context and fixes a taxonomy is created based on the review results.

11.4 CWE

Common Weakness Enumeration (CWE) [126] provides categories for software vulnerabilities. CWE-79 is the basic enumeration for Cross Site Scripting. It is distinguished between reflected XSS, stored XSS and dom-based XSS. In a developer perspective these are different sources and sinks depending on when the sanitization should happen. For example, if only sanitized data should be stored in the database, the category can be seen as sink category. On the contrary it can be seen as a source category, if sanitization should only occur before the data is presented on the web page. The CWEs 81 to 87 are different variants of the base CWE-79. These can be seen as different categories for failed sanitization methods and different HTML contexts. The table 11.2 shows source code pattern categories of the different CWE enumerations.

CWE-81	Improper Neutralization of Script in an Error Message Web Page	HTML context
CWE-82	Improper Neutralization of Script in Attributes of IMG Tags in a Web Page	HTML context
CWE-83	Improper Neutralization of Script in Attributes in a Web Page	HTML context
CWE-84	Improper Neutralization of Encoded URI Schemes in a Web Page	HTML context
CWE-85	Doubled Character XSS Manipulations	Failed sanitization
CWE-86	Improper Neutralization of Invalid Characters in Identifiers in Web Pages	Failed sanitization
CWE-87	Improper Neutralization of Alternate XSS Syntax	Failed sanitization

Table 11.2: CWE Cross Site Scripting variants mapped to categories.

11.5 Taxonomy of Source Code Patterns

In this section the different taxonomies resulting from the review process are described in detail. The correlations to more or less corresponding CWE categories are mentioned.

11.5.1 Sources

Figure 11.1 shows three categories for the sources. There are 51 sources because CVE-2014-9270 is a stored and reflected Cross Site Scripting vulnerability. Accordingly, for each type a different source was found (database source and user input source).

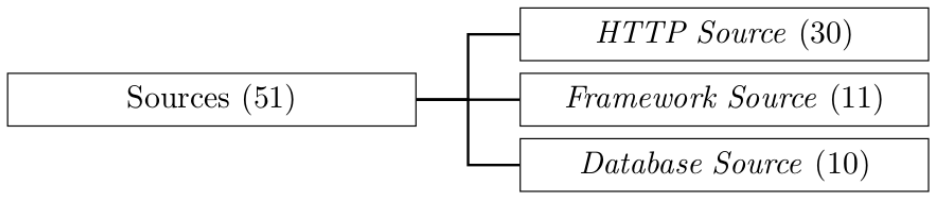


Figure 11.1: Taxonomy of sources based on the data set.

HTTP Source

Sources which fall in the *HTTP Source* category are basic HTTP methods provided by PHP. No wrapper was used which might trick developers into thinking that the data might already be sanitized. Sources from this category are related to reflected Cross Site Scripting vulnerabilities. In our data set we found *$_GET*, *$_REQUEST*, *$_POST*, *$_SERVER* and *$_COOKIE*. These are reserved variables from PHP [130]. *$_FILES*, *$_HTTP_RAW_POST_DATA* would also fall into this category, but these were not found in our data set.

Framework Source

PHP is commonly used with frameworks which provide features to create web pages more conveniently. Methods that wrap around methods from the *HTTP Source* category or methods provided from frameworks to get user provided data fall into this category (e.g. *gpc_get_string()*, *getParam()*). All our samples were internally using sources from the *HTTP Source* category. Accordingly, vulnerabilities with sources from this category are also related to reflected Cross Site Scripting. The extra category was created because such methods might already sanitized the input.

Database Source

Stored Cross Site Scripting stores the malicious input in the database. That input will be later on presented without any further sanitization on a web page. For our research the methods where user data is returned from the database is seen as a source. For example, in our data set we found functions like *db_fetch_row()* and *serendipity_db_query()*.

11.5.2 Insufficient Sanitization

There is a difference between sanitization methods and escaping methods. Sanitization methods are removing suspicious character which is commonly used to mitigate SQL injection vulnerabilities. In contrast escaping methods are just escaping suspicious character that cannot be used to inject any code. The resulting data from both methods will not be harmful anymore. Escaping methods are more commonly used to protect against Cross Site Scripting attacks because it will not remove characters that an user wants to be displayed. We group both methods together and call them sanitization methods which can either be sanitization methods or escaping methods. Most of our vulnerable samples did not use any sanitization methods. Nevertheless,

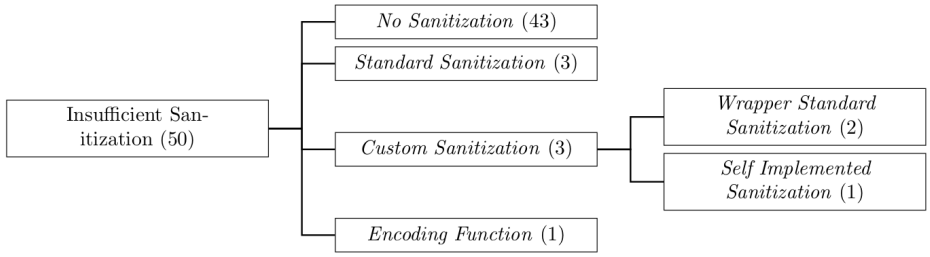


Figure 11.2: Taxonomy of insufficient sanitization methods based on the data set.

some did use well known sanitization methods, but Cross Site Scripting vulnerabilities still occurred. Figure 11.2 shows an overview of the categories for insufficient sanitization methods.

No Sanitization

As the name already indicates, no sanitization method was used. Samples where plain user input will reach a sink fall into this category.

Standard Sanitization

Developers actually used official sanitization methods like *htmlspecialchars()*, but a Cross Site Scripting vulnerability still existed. Why was that sanitization insufficient? Two of the three found samples, the sanitization was insufficient because the HTML sink was from the category *JavaScript Context*. If sinks are from that category, the sanitization has to be more specialized because the context is already in Javascript. How to prevent XSS attacks in such a context is described later on. The last sample did have a special condition where the sanitization method is not used. That sample is described in detail in section 11.6.

Custom Sanitization

Custom sanitization methods were also found in the data set. Two samples did use wrapper methods which internally use methods from the *Standard Sanitization* category. Accordingly, they should be protected against XSS attacks, but the sinks are again from the *JavaScript Context* category.

One sample did implement a sanitization method using *str_ replace()* method. The implementation was insufficient and the patch did fix the

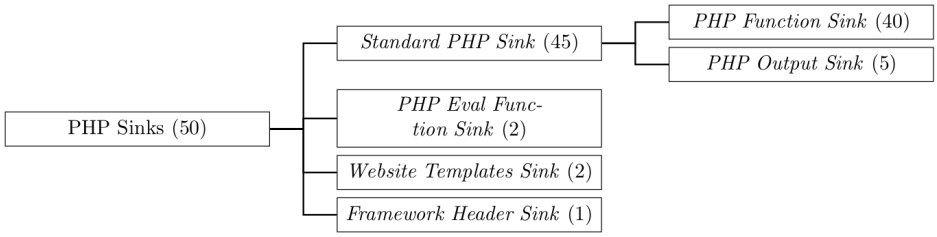


Figure 11.3: Taxonomy of PHP sinks based on the data set.

vulnerability by using a sanitization method from the category *Standard Sanitization*.

Encoding Function

One sample did use an encoding function. These functions are not supposed to be used as a sanitization method. Nevertheless, as seen in the fixes categories, some developers patched the vulnerabilities by using encoding functions. This just prevents Cross Site Scripting vulnerability as long the context is from the *Plain HTML Context* category and the correct HTML encoding function is used.

11.5.3 PHP Sinks

The sink categories are split into PHP and HTML sinks. An overview of the PHP sink categories is shown in figure 11.3. This sinks taxonomy is useful for developers because these are the sinks where user input without sanitization will be harmful.

Standard PHP Sink

This category covers standard output to the web page. Most of our data samples did have sinks in the *Standard PHP Sink* category. It is split into the first category *PHP Function Sink* where methods are used to output the data. Common methods are *echo()* and *print()*. The other category *PHP Output Sink* covers simple output in PHP files, where the *<?...?>* element is used.

PHP Eval Function Sink

In our data set two samples did have an *eval()* function as sink. These CVE reports were marked as Cross Site Scripting vulnerability. Actually these

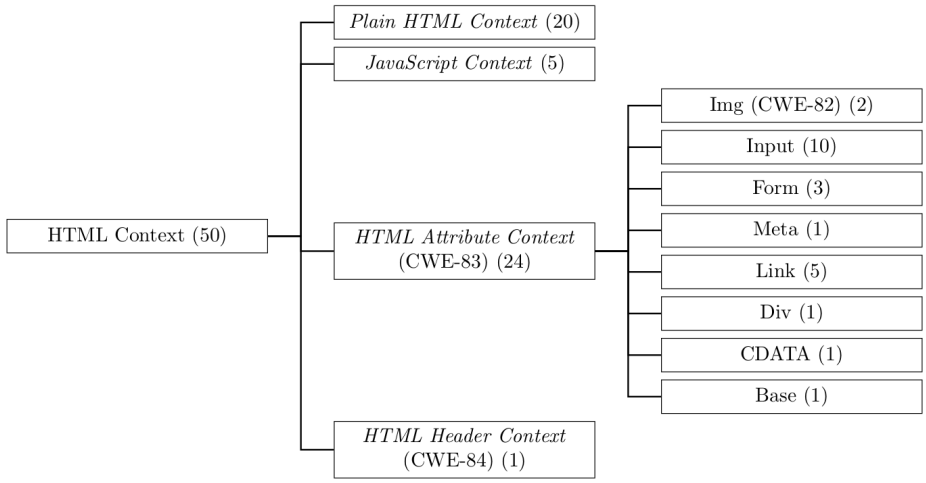


Figure 11.4: Taxonomy of HTML sinks based on the data set.

sinks also open up Direct Dynamic Code Evaluation vulnerabilities (CWE-95). Nevertheless, it can also result in a Cross Site Scripting vulnerability.

Website Templates Sink

Some PHP frameworks provide template files, which can be used to write PHP similar code with some extra features. In our data set two samples are using *tlp* files from the Smarty framework [132]. If such a framework template is used it falls into this category.

Framework Header Sink

One sample did have the sink in a header parameter. The PHP framework used in that sample provides a function to set a header parameter. Accordingly, this category was created for sinks which allow to modify the HTTP header.

11.5.4 HTML Context

The PHP sinks categories are more focused on what functions are used to print the data. HTML context rather focus on where the data is presented in the web page. This taxonomy is more similar to the categories provided by CWE. Figure 11.4 shows the categories found in our data set.

Plain HTML Context

Outputs which get into a context of this category are in a simple plain HTML part. No special condition like being in a Javascript context or being an attribute. Standard sanitization methods are well suited for such a context. Also the use of HTML encoding functions is enough as long the output is inside the HTML body [129].

JavaScript Context

Sinks where the output will be in a Javascript context fall into this category. Accordingly, sanitization must be more specialized. The OWASP XSS prevention sheet [129] explains that simple encoding functions are not enough. The output also should be quoted and sanitized to prevent any Cross Site Scripting attacks [129]. It is important to know that inputs must be data only. Otherwise, the prevention of Cross Site Scripting will be very difficult and requires further restrictions. In our data set only data was used inside a script tag. Another pitfall in this context is the method *htmlspecialchars()* because it does not remove simple quotes without setting the *ENT_QUOTES* parameter. If developers use simple quotes as escaping and do not set the parameter, XSS attacks are still possible.

HTML Attribute Context

This is the same category as CWE-83. The output is inside a HTML attribute. The CWE-82 is specialized version of being an image attribute. In our data set only two samples actually were in a image attribute context. The input has to be sanitized and quoted like in the *JavaScript Context* category.

HTML Header Context

One sample did have a sink from the category *Framework Header Sink*. Accordingly, the context is a HTTP header and that sample falls into this context category. To prevent any Cross Site Scripting attacks in a header, sanitization methods from *Standard Sanitization* are required. Encoding function will not be sufficient.

Sanitization in different contexts

As already mentioned, different HTML context require different sanitization methods. The table 11.3 provides an overview of what sanitization methods are sufficient enough to prevent any Cross Site Scripting attacks. Accordingly,

HTML Context	Encoding	Sanitization	Sanitization & Escaping
Plain HTML Context	prevent	prevent	prevent
HTML Attribute Context	insufficient	insufficient	prevent
HTML Header Context	insufficient	prevent	prevent
JavaScript Context	insufficient	insufficient	prevent

Table 11.3: CWE Cross Site Scripting variants mapped to categories.

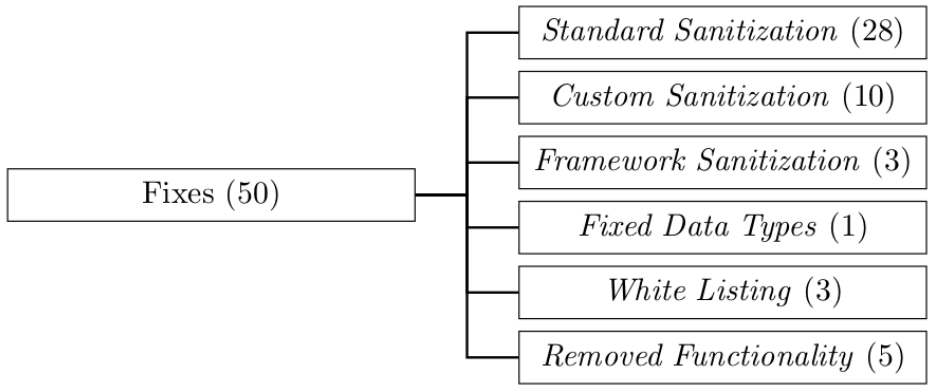


Figure 11.5: Taxonomy of fixes based on the data set.

it would be helpful to sanitize and escape any user input to be protected against XSS attacks in all HTML contexts.

11.5.5 Fixes

An important part to prevent Cross Site Scripting is correct sanitization of user input. Most reviewed projects had no sanitization before the patch related to the CVE report was developed. The patches contain different fixes to sanitize inputs. Figure 11.5 shows the taxonomy created for the fixes which were used in our data set.

Standard Sanitization

Fixes of this category used the standard functions provided by PHP. No combination of multiple sanitization methods or a sanitization method from a framework is used.

In our sample, common methods were *htmlentities()*, *htmlspecialchars()*, *urlencode()*, etc.

Custom Sanitization

Some patches did fix the vulnerability by using a custom developed sanitization method. Few used a combination of different sanitization methods from the category *Standard Sanitization* as sanitization method. Also some samples did fix the vulnerability by using a sanitization method which did use replace methods like *preg_ replace()*.

Framework Sanitization

Three samples were using sanitization methods provided by a framework to fix the vulnerability. Fixes that use a framework sanitization method falls into this category. For example, the *esc_ html()* function from WordPress [133] fall into this category.

Fixed Data Types

One sample did use a fixed data type to prevent any Cross Site Scripting vulnerabilities. An ID value was evaluated as an integer value. Accordingly, a simple and elegant way of fixing the vulnerability.

White Listing

White listing is a common way to prevent any injection attacks. Just fixed values are allowed and all other inputs will be ignored. Three of the samples used white listing to fix the vulnerabilities.

Removed Functionality

Another category to fix a vulnerability is to remove that output. Even some sample did remove some functionality which contained the vulnerability. At least it fixed the vulnerability.

11.6 Special case: CVE-2012-5163

To get a better understanding of the manual review process, this section provides an example of the CVE-2012-5163 report in the open source project Osclass [128]. Looking into the report notes reveals that the id parameter in *enable_ category* can be used to inject arbitrary code. Accordingly, the source is already known. Looking into the vulnerable source code reveals that the function *Params::getParam("id")* is used which falls in the ***Framework Source*** source category. The code snippet 11.6 shows the related source

code. As seen on line 9 and 14 standard sanitization methods are used. Consequently, it falls in the insufficient sanitization category ***Standard Sanitization***. Nevertheless, because *$htmlencode* is on default set to false, the sanitization method is not used. The function will return the value without any sanitization for the *enable_ category* id. As this sample shows, a developer might think that the *getParam()* function does already sanitize the inputs but in specific conditions it does not.

The code snippet 11.7 shows a shortened version of the *doModel()* method, which prints the not sanitized user input. Line 4 and 6 shows a little part of the data flow that was tracked by the manual review. The final PHP sink is found on line 7, where the output will be encoded by the function *json_ encode* and the sink is the *echo* function. Accordingly, as PHP sink was found from the ***Standard PHP Sink*** category. The *doModel()* functions creates a web page where the found echo result in a plain HTML context. Therefore it falls into the ***Plain HTML Context*** category. The fix was very simple by encapsulating the related *getParam()* call with a *strip_ tags()* functions. Thus the fix category is ***Standard Sanitization*** because it is a function provided by PHP. The encoding function is supposed to be used for a JSON context. Accordingly, this does not prevent any XSS attacks in a HTML context.

```
static function getParam($param, $htmlencode = false)
{
    if ($param == "") return '' ;
    if (!isset($_REQUEST[$param])) return '' ;

    $value = $_REQUEST[$param];
    if (!is_array($value)) {
        if ($htmlencode) {
            return htmlspecialchars(stripslashes($value),
                ENT_QUOTES);
        }
    }

    if(get_magic_quotes_gpc()) {
        $value = strip_slashes_extended($value);
    }

    return ($value);
}
```

Figure 11.6: Params::getParams() function with insufficient sanitization from CVE-2012-5163.

```
// root category
if( $aCategory['fk_i_parent_id'] == '' ) {
...
$aUpdated[] = array('id' => $id) ;
...
$result['affectedIds'] = $aUpdated ;
echo json_encode($result) ;
break ;
}
...
break ;
```

Figure 11.7: A shortened version of the sink source code part from CVE-2012-5163.

11.7 Discussion

The results did show that Cross Site Scripting vulnerabilities have a high rate of vulnerabilities where no sanitization is used. The vulnerable source code for CVE-2013-0807, CVE-2014-8793 and CVE-2013-4880 reports have the vulnerabilities including source and sink in one line of code. These results show that Cross Site Scripting is not as present in developers' minds as it should be. For Cross Site Scripting the different HTML contexts are relevant because as described in section 11.5.4 they require different sanitization methods. Accordingly, the prevention mechanisms and context categories have to fit to prevent any attacks.

The CWE sub categories for Cross Site Scripting are not very detailed. As our results show many different categories exist. Especially the context category *JavaScript Context* should exist because it requires more specialized sanitization to prevent any XSS attacks. The method *htmlspecialchars()* should probably also escape simple quotes as default because developers might not read the documentation carefully enough to know that an additional parameter is required to escape simple quotes. In a *JavaScript Context*context it opens up unnecessary XSS vulnerabilities.

11.8 Conclusion and Future Work

We analysed the source code of 50 GitHub projects which are correlated to CVE reports mentioning Cross Site Scripting and available source code patches. The results show different taxonomies for important source code patterns. Relations to the existing CWE categories are created. Our taxonomy is more focused on the developers point of view. In combination with our previous work [145], [146], three taxonomies for different vulnerabilities categories are created. These taxonomies allow further research using the taxonomy and the source code samples as a dataset. These categories can be used to get a better understanding where Cross Site Scripting vulnerabilities can occur. Especially, the HTML context taxonomy has a big influence on what prevention mechanisms should be used.

The sample set was very small; more samples should be analysed for the source code patterns and compared to our results. Another interesting aspect would be to research source code patterns of XSS samples in the programming language JavaScript. These patterns could also be compared to our results. This taxonomy could be used to improve the teaching of software security skills for developers. The knowledge of source code patterns can be used to create exercises. An interesting point will be to create these exercises

automatically similar to previous research projects [144] [134]. Existing source code from projects can be used and transformed to create these source code patterns. Another interesting point will be using these categories to test static code analysis tools. These can be investigated whether they detect all permutations. It will be interesting to see what combinations of the categories are difficult to be detected from static code analysis tools.

References

[124] CVE Common Vulnerabilities and Exposures. https://cve.mitre.org/.

[125] GitHub. https://github.com/.

[126] CWE - Common Weakness Enumeration. http://cwe.mitre.org/, 2015.

[127] OWASP Top Ten Project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2015.

[128] Osclass - open source classified. https://osclass.org/, 2018.

[129] XSS (Cross Site Scripting) Prevention Cheat Sheet. https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet, 2018.

[130] PHP - reserved variables. http://php.net/manual/en/reserved.variables.php, 2018.

[131] SAMATE - SARD, 2018. URL https://samate.nist.gov/SARD.

[132] Smarty PHP Template Engine. https://www.smarty.net/, 2018.

[133] WordPress. https://wordpress.org/, 2018.

[134] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. LAVA: Large-Scale Automated Vulnerability Addition. *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, pages 110–121, 2016. doi: 10.1109/SP.2016.15.

[135] M. V. Gundy, M. V. Gundy, H. Chen, and H. Chen. Noncespaces: using randomization to enforce information ow tracking and thwart cross-site scripting attacks. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 1–18, 2009.

[136] S. Gupta and B. B. Gupta. Js-san: defense mechanism for html5-based web applications against javascript code injection vulnerabilities. *Security and Communication Networks*, 9(11):1477–1495, 2016.

[137] Z. Hui, S. Huang, B. Hu, and Z. Ren. A taxonomy of software security defects for SST. *Proceedings - 2010 International Conference on Intelligent Computing and Integrated Systems, ICISS2010*, pages 99–103, 2010. doi: 10.1109/ICISS.2010.5656736.

[138] P. Lerthathairat and N. Prompoon. An approach for source code classification to enhance maintainability. *Proceedings of the 2011 8th International Joint Conference on Computer Science and Software Engineering, JCSSE 2011*, pages 319–324, 2011. doi: 10.1109/JCSSE.2011.5930141.

[139] M. T. Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. pages 331–346, May 2009. ISSN 1081-6011. doi: 10.1109/SP.2009.33.

[140] F. Massacci and V. H. Nguyen. Which is the right source for vulnerability studies?: An empirical analysis on Mozilla Firefox. *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, pages 4:1—-4:8, 2010. doi: http://doi.acm.org/10.1145/1853919.1853925.

[141] S. Maurya. Positive security model based server-side solution for prevention of cross-site scripting attacks. *12th IEEE International Conference Electronics, Energy, Environment, Communication, Computer, Control: (E3-C3), INDICON 2015*, pages 1–5, 2016. doi: 10.1109/INDICON.2015.7443473.

[142] S. Maurya and A. Singhrova. Cross Site Scripting Vulnerabilities and Defences: A Review. *International Journal of Computer Techonology and Applications*, 6(June):478 – 482, 2015.

[143] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. *Proceedings of the Network and Distributed System Security Symposium*, pages 1–42, 2009.

[144] F. Schuckert. PT: Generating Security Vulnerabilities in Source Code. In M. Meier, D. Reinhardt, and S. Wendzel, editors, *Sicherheit 2016 - Sicherheit, Schutz und Zuverlässigkeit*, pages 177–182, Bonn, 2016. Gesellschaft für Informatik e.V.

[145] F. Schuckert, B. Katt, and H. Langweg. Source Code Patterns of SQL Injection Vulnerabilities. *International Conference on Availability, Reliability and Security*, 2017. doi: 10.1145/3098954.3103173.

[146] F. Schuckert, M. Hildner, B. Katt, and H. Langweg. Source Code Patterns of Buffer Overflow Vulnerabilities in Firefox. *Proceedings of Sicherheit 2018*, pages 107–118, 2018. doi: 10.18420/sicherheit2018_08.

[147] B. Stivalet and E. Fong. Large Scale Generation of Complex and Faulty PHP Test Cases. *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, pages 409–415, 2016.

[148] Y. Wu, H. Siy, and R. Gandhi. Empirical results on the study of software vulnerabilities. *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 964–967, 2011.

Chapter 12

Difficult XSS Code Patterns for Static Code Analysis Tools

12.1 Introduction

Static code analysis tools exist for a long time now. Nevertheless, looking into OWASP top 10 and CWE top 25 lists shows that the same vulnerability types occur all the time. Are modern static code analysis tools sufficient to detect recent vulnerabilities which occurred in recent years? Another problem might be that the tools have too many false positive reports that discourage developers from using them? Are there patterns that are not detectable with state of the art static code analysis tools? False positives are vulnerability reports that are actually no vulnerabilities. Accordingly, these patterns should either not be mitigated by developers or should be taught to developers that they know what are critical parts for manual security reviews. Our goal of this study is to find problems and limitations of static code analysis tools. Is it possible to create small source code samples which have a high probability to create false positive or false negative reports? False negatives are vulnerabilities that are not reported by the tool. We focus on cross site scripting vulnerabilities because it is a widespread vulnerability. Source code from open source projects is used to find false positives and false negatives. These reports are then reviewed to find the corresponding problematic source code patterns. These patterns are evaluated using the initial static code analysis tools and two open source tools.

12.2 Related Work

Many research projects exist that evaluate static code analysers. Goseva-Popstojanova and Perhinschi [162] evaluated three commercial static code analysis tools. They calculated a G-score that used the detection and the false positive rate. This score was used to compare the different static code analysis tools. They scanned the Juliet database [149] to evaluate different permutations of security issues. They used three open source projects with known vulnerabilities for their evaluation. These projects were chosen based on CVE reports. The research showed that none of the tools detected all security issues. They stated that relying only on static code analysis for detecting security vulnerabilities will leave a large number of vulnerabilities undiscovered (false negatives). Our research also uses open source projects with known vulnerabilities. Instead of just scanning these projects to evaluate the tools, we identified different source code patterns based on the source code that are problematic for the tools.

Delaitre et al. [160] researched about effectiveness measurement from static code analysis tools based on significance, ground truth and relevance metrics. They used three different types of source code for their data set. The production software type did not have any known security vulnerabilities. Then they used source code related to CVE reports as their second type. Their last type was small samples where small source code samples were created specifically. They used these data sets to evaluate 14 static code analysis tools and presented the results from four tools. The results showed that the tools detect specific issues, but not a single tool detected all kinds of issues. Our work has very similar results. None of our evaluated tools could detect all vulnerabilities from our created data sets.

The research from Díaz and Bermejo [161] evaluated different open source and commercial static code analysis tools. They focused on the detection rate and the false positive rate. They used the SAMATE database to evaluate the tools. From the SAMATE database they used the test suite 45 that contains security vulnerabilities and the test suite 46 that has the vulnerabilities fixed. For the evaluation the F-measure from Van Rijsbergen [165] was calculated.

Another article from AlBreiki and Mahmoud [158] evaluated three open source tools. The tools have different approaches. One tool analyzes source code (OWASP Yasca), the next one analyzes byte-code (FindBugs) and the last one analyzes binary code (Microsoft Code Analysis Tool .NET). The evaluation test cases are based on top security issues from OWASP/CWE/SANS. False positives were not included in the evaluation. Basso et al. [159] researched about how software fault injection will affect static code analysis

tools. They showed that software fault injection affects the detection rate of the tested static code analysis tools. Primarily, it created false positive reports and the existing vulnerabilities were not detected. Zhioua et al. [166] evaluated four static code analysis tools. They described how security issues and security properties are related to each other. Their focus is more on how the tools detect vulnerabilities and what techniques the tools use. Khare et al. [163] also evaluated static code analysis tools. Their approach was based on large samples with more than 10 million lines of code. Their test showed that less than 10% of vulnerabilities were detected. Even some simple vulnerabilities were not detected.

12.3 Methodology

We use multiple static code analysis tools to find and evaluate problematic source code patterns. Problematic patterns result in false negative and false positive reports. This section provides background knowledge, explains what commercial and open source static code analysis tools were chosen and how the problematic patterns are examined.

12.3.1 Background

False negative (FN) reports mean that a tool did not report a vulnerability that actually exists. To find source code patterns that create such false negative reports, initial source code is required that contains such patterns. In contrast false positive (FP) reports mean that a tool reports a vulnerability that actually is no vulnerability. To find such patterns, source code is required that might look like there is a vulnerability, but one can be sure that the program is secure.

12.3.2 Selected static code analysis tools and data set

We selected three commercial static code analysis tools. These tools are called Tool A, Tool B and Tool C. Our licences do not allow to publish the name of the tools. Our primary goal is not to evaluate the tools against each other. Instead we want to identify source code patterns that are problematic for state of the art static code analysis tools. All of these tools support data flow analysis and use it for their analysis. The tools were all up to date when the scans were started in September 2018.

For the evaluation of specific patterns two open source static code analysis tools (Exakat [152] and Sonarcloud [157]) were added. To find relevant open source static code analysis tools a collection of open source static code analysis

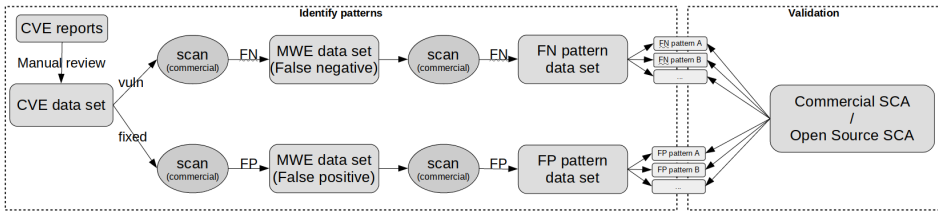


Figure 12.1: XSS Vulnerability analysis process.

tools [150] was reviewed. First of all, all tools that do not scan for security issues or are not maintained anymore (last update $>$ 1 year) were filtered out. The remaining tools were checked, if they can detect cross site scripting vulnerabilities. Only Exakat and Sonarcloud were found to be suitable for the validation.

We used the CVE data set from the work [164]. These samples are source code from open source projects that are related to CVE reports from 2010 until 2016 (seven years) focusing on cross site scripting vulnerabilities. Those reports are old enough to provide the static code analysis tool developers enough time to adapt their algorithms. We use the corresponding vulnerable and fixed versions. The data set contains 50 vulnerable and 50 fixed samples.

12.3.3 Vulnerability analysis

The vulnerability analysis process is shown in figure 12.1. The commercial tools are used to scan the related source code. Because of previous investigation we already know where the corresponding sinks are located. This information is used to **automatically check** if the vulnerability is not found (false negative) or if the fixed version still is reported as a vulnerability (false positive). Because our previous work was a manual review, the false positive and false negative reports are checked if these are really false positive or false negative reports. After that check all interesting samples are identified and manually reviewed to find the problematic source code patterns.

The **manual review** is done by checking the data flow from the source to the corresponding sink. A source is a function or method where data is provided by a user. Such data can be potentially dangerous if it reaches critical functions. Such critical functions are called sinks. It is reviewed for code patterns which might be problematic for static code analysis tools. Because each of the reviewed samples are either a false positive or a false negative report, there must be something in the data flow which might interrupt the data flow algorithm or confuses the static code analysis algorithm. Based on

the review results a minimal working example (MWE) data set is created that contains simple source code samples that imitate the vulnerabilities. The corresponding sink is noted down for further scans. To ensure that the samples are still vulnerable, exploits are written that result in a cross site scripting vulnerability.

The next step is to check if problematic source code patterns are found. The simple samples are **scanned again** by the three commercial static code analysis tools. Again the results can be automatically checked because the corresponding sinks are known. All simple samples that created false positive and false negative reports are then designated as problematic patterns for static code analysis tools. These samples are in the FN pattern and FP pattern data set. The results from the scans and the identified problematic patterns are presented in this article.

Because these samples can contain multiple source code patterns at once, further scans are required to **validate** which of these patterns are problematic. For each source code pattern a specific sample is created that just contains that pattern. These samples are scanned by the commercial and open source tools again to evaluate which of these patterns are problematic.

12.4 Static code analysis results

As described in the previous chapter, our data set for the first scan contains source code of 50 vulnerable and 50 fixed projects. For this work that data set is labeled as "CVE data set" because it is the source code related to CVE reports. Accordingly, each tool scanned 100 projects. For our work only false positive and false negative reports are important. Figure 12.2 shows how many false positive and false negative reports the three tools issued on the CVE data set. It shows that tool C has a lot more false positive reports but it has the lowest false negative reports. This is an issue static code analysis tools have, if they want to detect more vulnerabilities this usually also means more false positive reports. Accordingly, tool C is more noisy than tools A and B. Investigating the false reports of tool C shows that common sanitization methods like *htmlspecialchars()* are reported as a possible cross site scripting vulnerability. Tool C rates reports with such sanitization functions not as critical as without such functions. Nevertheless, if such functions are used in the correct context, they are sufficient to prevent cross site scripting attacks. Depending on the context it also might not be sufficient. The tools have to check the context of the sink to successfully decide if it is a vulnerability or not. In the CVE data set we assume all of the fixed samples use sufficient sanitization methods for their context. Table 12.1

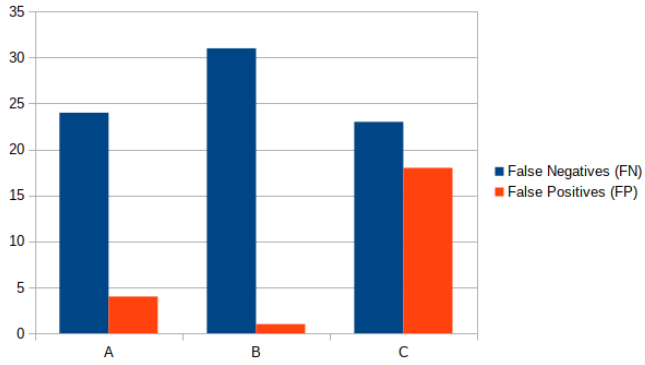


Figure 12.2: False positive and false negatives reports on the CVE data set.

Table 12.1: Overview of metrics for the commercial tools.

Metric	Tool A	Tool B	Tool C
Number of projects	50	50	50
True Positive	26	19	27
False Negative	24	31	23
Number of fixed	50	50	50
True Negative	46	49	32
False Positive	4	1	18
Accuracy	72%	68%	59%
Recall	52%	38%	54%
False Alarm Prob.	8%	2%	36%

shows the computed metrics of the results. As stated, tool C has a higher probability of a false alarm (false report). If we look into these metrics the recall rate indicates how many of the existing vulnerabilities will be reported. Only one tool is below 50%. Nevertheless, there are still vulnerabilities that are not detected by the other tools as well. Also the accuracy metric results are in a good range. The accuracy shows how accurate the reports are related to be a true positive or true negative report. These metrics are interesting to see, but for a comparison our data set does not contain enough samples. From the vulnerable CVE samples there were 35 samples that got from at least one tool a false negative report. In the fixed CVE samples, 18 samples produced a false positive report from at least one tool.

12.5 Minimal working example data set

We reviewed the source code of the false positive and false negative reports from the CVE data set. We created a data set that imitates the source code with minimal required lines of code. Some samples of the CVE data set contained the same relevant source code patterns. We only created one sample if the same source code patterns were found because the minimal working examples would be the same. In this work that data set is called minimal working example "MWE data set". The data set contains 25 false negative samples and 12 false positive samples. Accordingly, 13 vulnerable samples from the CVE data set and 6 out of the fixed samples from the CVE data set were duplicates. A reason for this was that, some CVE reports are from the same open source project but different versions and they used similar code parts. Table 12.2 shows results of the scan on the samples where the expected outcome should be a false negative report. Five samples did not create a false negative report because the count of false negative reports were zero. Accordingly, the relevant source code patterns were not examined for these 5 samples. Nevertheless, 20 samples created a false negative report. Table 12.3 shows the results of the scan of the samples where the expected outcome should be false positive reports. Ten out of twelve samples provoked false positive reports. The resulting source code patterns for false positive and false negative reports are presented in section 12.7.

Table 12.2: Scan results from false negative MWE data set with 25 samples.

CVE related	Patterns	Tool A	Tool B	Tool C	Count of FN	Type
CVE-2015-7777	$_SERVER	✓	FN	✓	1	reflected
CVE-2014-3544	Inheritance DB Implementation Class variable assignment by string SQL DAO List assignment - list()	FN	FN	FN	3	stored
CVE-2015-1347	-	✓	✓	✓	0	reflected
CVE-2011-2938	Unc. function call - call_user_func_array Unserialize	FN	FN	FN	3	stored
CVE-2015-1562	-	✓	✓	✓	0	reflected
CVE-2012-2331	*Foreach* on super global variable Scope global Return value by reference	✓	FN	✓	1	reflected
CVE-2012-5608	Sink print_r	FN	FN	✓	2	reflected
CVE-2013-7275	Template XML file	FN	FN	FN	3	reflected
CVE-2014-3774	-	✓	✓	✓	0	reflected
CVE-2014-4954	DB wrapper List assignment - list()	FN	FN	FN	3	stored
CVE-2014-9270	SQL DAO	✓	FN	✓	1	stored
CVE-2015-5076	-	✓	✓	✓	0	reflected
CVE-2011-3358	Unc. function call - call_user_func_array	FN	FN	FN	3	reflected
CVE-2012-5339	DB Source Unc. function call - string	✓	FN	FN	2	stored
CVE-2011-4814	Conditional sanitization - disabled	✓	FN	✓	1	reflected
CVE-2013-1937	Input valid type check	FN	FN	✓	2	reflected
CVE-2012-2129	Keyword global	FN	FN	✓	2	reflected
CVE-2012-4395	Template output buffering	FN	FN	FN	3	reflected
CVE-2014-9219	Input valid type check	FN	FN	✓	2	reflected
CVE-2015-7348	Replacement	✓	FN	✓	1	reflected
CVE-2014-9271	Sink - header	FN	FN	FN	3	stored
CVE-2013-0807	-	✓	✓	✓	0	reflected
CVE-2014-9281	Stripslashes	✓	FN	✓	1	reflected
CVE-2014-9269	$_COOKIE Stripslashes	✓	FN	✓	1	reflected
CVE-2012-5163	Conditional sanitization - disabled	FN	FN	✓	2	reflected
Summary (FN)		12	20	10		

Table 12.3: Scan results from false positive MWE data set with 12 samples.

CVE related	Patterns	Tool A	Tool B	Tool C	Count of FP	Type
CVE-2013-0201	Standard - specialchars	✓	✓	FP	1	reflected
CVE-2011-4814	Custom - preg_replace	FP	✓	FP	2	reflected
CVE-2014-8352	Custom - reset if not valid	✓	FP	FP	2	reflected
CVE-2013-0275	Standard - specialchars	✓	✓	FP	1	reflected
CVE-2014-9571	Standard - specialchars Custom - preg_replace	✓	✓	FP	1	reflected
CVE-2015-5356	Standard - htmlentities	✓	✓	FP	1	reflected
CVE-2012-5163	Library - HTMLPurifier	✓	✓	FP	1	reflected
CVE-2011-2938	-	✓	✓	✓	0	stored
CVE-2014-9269	-	✓	✓	✓	0	reflected
CVE-2014-2570	Standard - htmlentities	✓	✓	FP	1	reflected
CVE-2013-4880	Standard - specialchars	✓	✓	FP	1	reflected
CVE-2011-3371	Standard - specialchars	✓	✓	FP	1	reflected
Summary (FN)		1	1	10		

12.6 Stored Cross Site Scripting

Cross site scripting vulnerabilities are categorized into reflected and stored. Static code analysis tool can simply find reflected cross site scripting vulnerability by doing a data flow analysis and see if tainted data will reach a possible sink. Tainted data is data can be manipulated by the user/attacker. Stored cross site scripting is a bit more difficult to be successfully detected.

Figure 12.3 shows an overview of stored cross site scripting. It is split into two main parts. The insert part (1,2,3) uses a common source like *$_GET*. Then a SQL statement will be created that will store the tainted data in the database. In another source code location a database source exists. These are functions that allow getting data from a database. A SQL statement will get the tainted data out of the database. Then it will be shown on a webpage. If no sanitization is used, a stored cross site scripting vulnerability would exist. As seen in figure 12.3, sanitization can be done in two different steps, (2) and (6). Either before inserting tainted data it can be sanitized (2) or it can be sanitized before the data will be displayed on the webpage (6). Static code analysis tools have to know the relation between the SQL statements (3) and (5) to successfully detect stored cross site scripting vulnerabilities without any false reports.

Samples were created to check if the tools can detect the different scenarios. Table 12.4 shows the result of the scans on the samples. The two open source tools were tested as well. Tool B and the open source tools did not even detect the simple stored cross site scripting sample, and consequently they will not report any vulnerabilities in the fixed versions. Tool A detects all samples correctly. Accordingly, it knows the relations between the INSERT and SELECT SQL statements. Tool C does at least detect the sample where the sanitization happens before the data reaches a sink. To detect stored

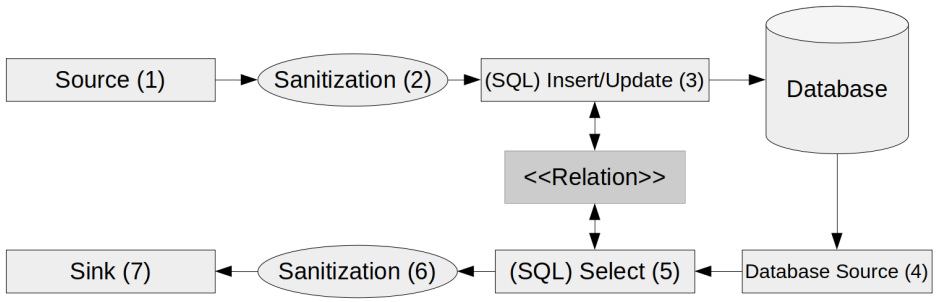


Figure 12.3: Overview of an stored XSS vulnerability.

Table 12.4: Scan results on the different stored and reflected XSS samples.

False negative pattern	Tool A	Tool B	Tool C	Exakat	Sonarcloud
Reflected XSS sample	✓	✓	✓	✓	✓
Stored XSS vulnerable	✓	FN	✓	FN	FN
Stored XSS sanitization insert (2)	✓	✓	FP	✓	✓
Stored XSS sanitization view (6)	✓	✓	✓	✓	✓

cross site scripting, the tools have to define database functions as a possible sources for tainted data. To reduce false positive reports, the relation between the SQL statements are important.

12.7 Difficult source code patterns

The review process required to look for suspected source code patterns which might be problematic for static code analysis tools. This section describes what suspected source code patterns were used to create the MWE data set. Only patterns are presented which are related to samples which created false positive or false negative reports.

12.7.1 Sources

Three sources were considered to be problematic for static code analysis tools.

$_SERVER

The super global variable *$_SERVER* is a special variable that contains some entries that are just controllable by the server. Nevertheless, it also has some entries which can be modified by the users. In our data *$_SERVER['PHP_SELF']* was used. This can be easily modified by inserting the payload in the url.

Foreach on super global variable

Most samples used super global variables by directly accessing the required key. For example, *$_GET['id']* is a commonly seen source. One sample did not access them directly instead it used a *foreach* loop to access all keys with the corresponding values.

$_COOKIE

One sample in our data set had the cookies as source. It uses the super global variable *$_COOKIE*.

12.7.2 Stored XSS Sources

As already described in section 12.6 stored cross site scripting vulnerabilities have two different sources. This section focus on the database sources and what source code patterns occurred in the data set.

Inheritance DB Implementation

One sample used an inheritance database implementation. That implementation does execute the query depending on the inherited database.

SQL DAO

Two samples used a database access object (DAO). Such an object stores the relevant information as class variables. It provides a function to create a SQL statement using the class variables as parameters.

DB wrapper

One sample used an database wrapper object. It wrapped the database object which is used to access the database. The wrapped object is depending on the configuration of the project. Different databases require different database objects. This samples used a *switch case* based on a String variable to choose the database implementation. The database wrapper is defined as a global variable.

12.7.3 Data flow

The main focus of this research is about source code patterns that might interrupt or confuse the data flow analysis. Nowadays the sources and sinks are well known. Accordingly, not many specific source or sinks are identified as problematic source code patterns. It is getting difficult for static code analysis tools to follow all kinds of data flows that are possible. This section will describe different source code patterns that cause problems for our evaluated static code analysis tools.

Unconventionally function call

The programming language PHP offers a lot of different features. One feature is to call functions in a unconventional way using other functions. Such functions allow to call a function that might not be known at the implementation time. In our CVE data set, multiple function call via the function *call_user_func_array* were found. Calling a function using a string variable was found in our data set as well. (E.g. *$functionName($param1, $param2)*)

Global variables

Global variables are very common in programming language. Also PHP allows to define global variables. This happens either by simply defining a variable in a PHP file without being inside a scope like a function or class. Another possibility is by defining a variable using the *global* keyword. This can be used inside a scope and is still a global variable.

Return value by reference

In the programming language C it is common to use function parameters to return values. This requires a copy by reference parameter. PHP also allows to pass a reference as a parameter. In our data set one samples used such copy by reference to pass the tainted data.

List assignment

Another language concept from PHP allows to assign values from a list to variables. This can either be used by using / / brackets or using the *list()* function.

Singleton

Another well known source code pattern is the singleton. This is a class where only one instance is allowed to exists. This is usually archived by defining the constructor of a class private. A static method has to be added which return the single instance of the class. This allows similar to a global variable to assign and access data from anywhere.

Class variable assignment by string

Similar to a function call, using a String allows PHP to assign class variables to determine which class variable should be used. For example, if you want

to assign the class variable with the name *foo* you can use following code:

```
1 $varName = 'foo';
2 $this->$varName = 'value';
```

Unserialize

A serialization framework is already included in PHP. Before PHP 7.1 the function *unserilize()* was very dangerous to use. Since PHP 7.1, it does only allow to unserialize an array or boolean [154]. Nevertheless, this function can be used to assign tainted values to an array.

12.7.4 Failed sanitization

Another way of provoking false negative report is by having functions that look like sanitization functions, but are insufficient or simply disabled by a conditional variable. The patterns we found in our samples are described in this section.

Conditional sanitization - disabled

Three samples had sanitization methods which were disabled by a conditional variable.

Input valid type check

Three samples had a method to check, if the input is a valid type. In our sample it was checked, if the input is an array. This does not prevent any cross site scripting issues.

Replacement

One sample used *str_ replace* for replacing some characters in the tainted string. This can also let the static code analysis tools think, that the input is sanitized.

Stripslashes

Two samples used the *stripslashes* function. This is insufficient depending on the context where the tainted data will be shown.

12.7.5 Sink

Two sinks were found which created a false negative report. One is the *header* function. Since PHP version 5.1.2, this is not considered as a sink anymore. Since then it was exploitable by inserting new line characters to do a header injection attack. Accordingly, this sink will not be relevant anymore as long up to date PHP is used. The other sink in the data set was the function *print_r* this is a function which prints data well formated. This is a exploitable sink for cross site scripting attacks.

12.7.6 Template

Modern web pages are commonly developed by using frameworks which are using templates. Different template implementations were found in our data set.

Template output buffering

Templates can be archived by using the output buffering feature. This allows to write the outputs from the template into a buffer. That buffer can be printed later on. This can result in a cross site scripting vulnerability.

Template XML file

Two samples used a XML file for the templates. A template file can be used to define multiple output pages. Each entry is put into a *CDATA* field. The PHP functions *simplexml_load_file* and *xpath* are used to get the related *CDATA* field. Then the *eval* function is used. This allows the templates to use variables like it was a normal PHP file. This also opens up more dangerous attacks as cross site scripting. But this is not the focus of this work.

12.7.7 Sanitization methods

Some false positive reports were found. This section will describe what patterns were found that sanitized the tainted input sufficient, but the static code analysis tools reported a vulnerability.

Standard

There are some standard sanitization methods which can be used to sanitize the tainted input. These are just sufficient enough, if the context of the

sink fits to the sanitization method. In our data set 7 samples used the *htmlspecialchars* and two samples used the *htmlentities* function.

Library - HTMLPurifier

One sample used the HTMLPurifier [153] library to defend any cross site scripting attacks. This prevents any attacks, but it is very difficult for static code analysis tools to know that such a library will sanitize against cross site scripting attacks.

Custom - preg_replace

One sample sanitized the input using the *preg_replace* function. This allows to successfully prevent any cross site scripting attacks. For example, all characters instead of numerics will be replaced with a whitespace. This prevents any cross site scripting attacks depending on the context. Static code analysis tools have to analyze the regular expression to validate, if the sanitization is sufficient.

Custom - reset if not valid

A simple solution to prevent against any attacks is to only allow numbers as inputs. This can be archived by casting a variable to integer. In one sample, the solution was to use the *is_numeric* function from PHP to check, if the input is numeric. If it is not numeric, the input will be reset to an empty string. Static code analysis tool have to check the condition and see that only numbers will be passed. This is another difficulty for static code analysis tools.

12.7.8 Validation

The different patterns were identified. Without further scans there would be no validation which of these patterns are really problematic. Specific samples were created which contain only one problematic pattern. The corresponding samples can be found in a GitHub repository [156]. To validate that the sink and source we used in our samples are known by the tools a simple XSS sample was created. It uses *$_GET* as source and *echo* as sink. Table 12.4 shows that each tool detected the simple reflected XSS vulnerability.

Table 12.5 shows the result for the patterns which expected to create false positive reports. The two open source static code analysis tools were added to the validation. Two samples were problematic for each of the five static code analysis tools. Of 25 patterns, one (Input valid type check - string)

Table 12.5: Scan results on the different false negative patterns.

False negative pattern	Tool A	Tool B	Tool C	Exakat	Sonarcloud	Count of FN	PHP Test suite
Class variable assignment by string	FN	✓	FN	FN	FN	4	-
SQL DAO	FN	FN	✓	FN	FN	4	-
Inheritance DB Implementation	FN	FN	✓	FN	FN	4	-
DB wrapper	FN	FN	FN	FN	FN	5	-
Conditional sanitization - disabled	✓	FN	✓	FN	FN	3	✓
Input valid type check - array	✓	FN	✓	FN	✓	2	-
Input valid type check - string	✓	✓	✓	✓	✓	0	-
Replacement	✓	FN	✓	✓	FN	2	-
Stripslashes	✓	FN	✓	✓	FN	2	-
Keyword global	✓	✓	✓	FN	FN	2	-
Scope global	✓	✓	✓	FN	FN	2	-
List assignment - brackets	✓	✓	✓	FN	FN	2	-
List assignment - list()	✓	✓	✓	FN	FN	3	-
Return value by reference	✓	FN	✓	FN	FN	3	-
Singleton	FN	✓	FN	FN	FN	4	-
Sink print_r	FN	✓	✓	✓	✓	1	-
$_COOKIE	✓	FN	✓	✓	✓	1	-
Foreach on super global variable	✓	✓	✓	FN	FN	2	-
$_SERVER	✓	FN	✓	FN	FN	3	-
Template output buffering	FN	✓	✓	FN	FN	3	-
Template XML file	FN	FN	FN	FN	FN	5	-
Unc. function call - call_user_func_array	✓	FN	FN	FN	FN	4	-
Unc. function call - call_user_func	✓	FN	FN	FN	FN	4	-
Unc. function call - string	✓	FN	FN	FN	FN	4	-
Unserialize	FN	FN	✓	FN	FN	4	✓
Summary (FN)	9	15	7	20	21		

did not result in any false negative reports. Accordingly, that pattern is not problematic for state of the art static code analysis tools. Six patterns were not problematic for the commercial tools. There are three patterns (SQL DAO, Inheritance DB Implementation, DB wrapper) related to a database source. As previous seen, the three tools Tool B, Exakat and Sonarcloud did not even detect samples with a simple database source. These tools cannot detect the more specific samples. The specified samples even tricked Tool A which performed best at the simple samples. Only Tool C detects two of these samples. This might be because they simply define database methods as possible sources without even checking if it is user provided data.

We looked up if our source code samples are already defined in the Common Weakness Enumeration [151] (CWE) data-base. For Cross Site Scripting there are no different patterns defined as CWE cases. We also looked up if our samples are present in the PHP Vulnerability Test Suite [155] from NIST. Table 12.5 shows that two of our samples were found in the test suite.

The false positive patterns were only five samples. Table 12.6 shows the scan results. The sample using the *HTMLPurifier* library did not create any false positive reports. It may either be that the tools saw the library as a sanitization method or the data flow was interrupted by the library. As

Table 12.6: Scan results on the different false positive patterns.

False positive pattern	Tool A	Tool B	Tool C	Exakat	Sonarcloud	Count of FP
Standard - htmlentities	✓	✓	FP	✓	✓	1
Library - HTMLPurifier	✓	✓	✓	✓	✓	0
Standard - specialchars	✓	✓	FP	✓	✓	1
Custom - preg_replace	✓	✓	FP	FP	✓	2
Custom - reset if not valid	✓	FP	FP	FP	✓	3
Summary (FP)	0	1	4	2	0	

already stated tool C reports cross site scripting vulnerabilities at a lower risk even when sanitize functions are correctly used. The *Custom - reset if not valid* sample also reported a false positive in tool B. Our samples were not able to produce a false positive report for tool A.

If a combination of the different reports is used, only the two problematic patterns (DB wrapper, Template XML file) would not be detected. On the other hand, that would also increase the false positive rate.

12.8 Discussion

Our results show that two commercial tools in our data set had an above 50% recall rate. Accordingly, if all projects related to the data set would have used one of the two tools it could have prevented at least 50% of the vulnerabilities. The tools are useful for software developers to get an idea where a vulnerability might be. Nevertheless, one cannot be sure that scanned source code is free of vulnerabilities, even if the tools did not report any vulnerabilities. Each tool has advantages and disadvantages, and all of them have problems with specific source code patterns. Some patterns were difficult for all of our tested tools. We cannot specify special patterns that could be declared as problematic for static code analysis tools because of technical reasons. All of the tools have specific patterns that are problematic for them. Also the open source tools did not perform better on the source code patterns. The static code analysis tools also allow some configurations for specific projects. For example, custom sanitization methods can be declared. This can help in some of our patterns, but most of our patterns are more based on programming language features like specific *if* conditions.

Some patterns are not the best practice in programming. For example, using a return value by reference. Modern tools should detect such coding variants. Nevertheless, developers could prevent such unconventional programming methods to make the static code analysis tools results more precise. On the contrary, the use of templates is very common and useful.

Tools have to be able to scan templates as well. In our data set there was one sample which used *tpl* files for templates. That sample was ignored because these templates use a custom language. Nevertheless, the samples using an output buffer that uses normal PHP files for the templates should be detected by all tools. The XML sample is bit more difficult because it uses a different file format and an *eval* function is used. Altogether developers can mitigate some patterns, but the static code analysis tools still can improve their algorithms.

Our researched patterns can be used as teaching examples. These are very interesting because these patterns are not detected by all static code analysis tools. Future work could research if today's learning examples cover such patterns. For example, do today's capture the flag events use such patterns or do they just use vulnerabilities that can be easily detected by static code analysis tools? Teaching the vulnerabilities that can be detected by tools is useful for basic understanding. Problematic patterns should be taught as advanced skill set. Our results show that focus on stored cross site scripting would be beneficial. Only one Tool A reported correctly the simple stored XSS samples and still had problems with our stored XSS source code patterns. Stored XSS is still difficult for state of the art static code analysis tools because of the required relation between the SQL statements.

Our data set is not very large, we used 50 samples with a vulnerable and patched version. The data set is not sufficient to compare tools against each other. It was sufficient to find some source code patterns which are problematic for static code analysis tools. Further research using a data set with other CVE reports might reveal more problematic patterns. Our patterns are also very specific to the static code analysis tools we used. For example, the *print_r* function which created a false negative report on tool A is probably just a missing sink in the analyzing part. It could be researched why these patterns are problematic for static code analysis tools. This would require access to the algorithm of the static code analysis tools or at least a documentation about how the algorithms work. We do not have access to such documentation. Our test was a black box test of static code analysis tools.

Our results do not reveal a lot of problematic patterns for false positive reports. One reason might be that tools have to detect a vulnerability correctly in the vulnerable sample. If they do not detect the vulnerability in the vulnerable version, it cannot create a false positive report in the patched version. This reduces the samples we could use for the review process.

12.9 Conclusion

Our goal was to find patterns that are problematic for static code analysis tools. The analysis of the source code from open source projects related to CVE reports revealed 19 source code patterns which led to false negative reports. The commercial tools provided better results than the open source tools. Nevertheless, the commercial tools also had problems with our identified patterns. Some can be mitigated by the software developers of the scanned projects. Others should be correctly detected by the static code analysis tools. The patterns can be used to improve static code analysis tools. For example, the patterns can be used as a test suite like the "PHP Vulnerability Test Suite" from SARD [155]. Overall, our results show that there are still a lot of source code patterns that are problematic for static code analysis tools. Developers who use static code analysis should know that there still might be undetectable vulnerabilities in their projects. Accordingly, developers should get taught those problematic patterns. Training of developers could be targeted to identifying and avoiding especially those patterns that are hard to flag by static analysis.

References

[149] Juliet Test Suite. `http://samate.nist.gov/SRD/testsuite.php`.

[150] PHP Static code analysis tools list. `https://github.com/exakat/php-static-analysis-tools`.

[151] CWE - Common Weakness Enumeration. `http://cwe.mitre.org/`, 2015.

[152] Exakat. `https://www.exakat.io/`, 2019.

[153] HTMLPurifier. `http://htmlpurifier.org/`, 2019.

[154] PHP manual. `https://www.php.net/manual/de/function.unserialize.php`, 2019.

[155] Software Assurance Reference Dataset Testsuite. `https://samate.nist.gov/SARD/testsuite.php`, 2019.

[156] Difficult Source Code Patterns. `https://github.com/fschuckert/sca\_patterns`, 2019.

[157] Sonarcloud. `https://sonarcloud.io`, 2019.

[158] H. H. AlBreiki and Q. H. Mahmoud. Evaluation of static analysis tools for software security. *2014 10th International Conference on Innovations in Information Technology (IIT)*, pages 93–98, 2014. doi: 10.1109/INNOVATIONS.2014.6987569.

[159] T. Basso, P. C. S. Fernandes, M. Jino, and R. Moraes. Analysis of the effect of Java software faults on security vulnerabilities and their detection by commercial web vulnerability scanner tool. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 150–155, 2010. doi: 10.1109/DSNW.2010.5542602.

[160] A. Delaitre, B. Stivalet, E. Fong, and V. Okun. Evaluating Bug Finders – Test and Measurement of Static Code Analyzers. *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)*, pages 14–20, 2015. doi: 10.1109/COUFLESS.2015.10.

[161] G. Díaz and J. R. Bermejo. Static analysis of source code security: Assessment of tools against SAMATE tests. *Information and Software Technology*, 55(8):1462–1476, 2013. ISSN 09505849. doi: 10.1016/j.infsof.2013.02.005.

[162] K. Goseva-Popstojanova and A. Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, 2015. ISSN 09505849.

[163] S. Khare, S. Saraswat, and S. Kumar. Static Program Analysis of Large Embedded Code Base: An Experience. *Proceedings of the 4th India Software Engineering Conference 2011*, pages 99–102, 2011.

[164] F. Schuckert, M. Hildner, B. Katt, and H. Langweg. Source code patterns of cross site scripting in php open source projects. In *Proceedings of the 11th Norwegian Information Security Conference*, 2018.

[165] C. J. Van Rijsbergen. *Information Retrieval, 2nd edition.* 1979.

[166] Z. Zhioua, S. Short, and Y. Roudier. Static Code Analysis for Software Security Verification: Problems and Approaches. In *Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International*, pages 102–109, 2014.

Chapter 13

Difficult SQLi Code Patterns for Static Code Analysis Tools

13.1 Introduction

Static code analysis tools are commonly used to find vulnerabilities in the development phase of a software project. These tools can be part of continuous integration to report potential security vulnerabilities before those reach a release version. Developers have to review these reports if those reports are actual vulnerabilities. If a reported vulnerability turns out not to be one, the report will produce unnecessary workload. Additionally, if a static code analysis tool does not report an actual vulnerability, vulnerabilities will have a high chance to be included in the release version of the product. Recent research [180] shows that there are source code patterns that are still difficult for static code analysis tools. It is important to identify such difficult patterns to mitigate them in the development phase or improve static code analysis tools to correctly handle them. If such difficult patterns are not known and these patterns are used, software may be developed and deployed with undetected vulnerabilities.

Our contribution is to answer the following research questions regarding SQL injection (SQLi) vulnerabilities:

1. What are difficult source code patterns for static code analysis tools?

2. Is it possible to create simple vulnerability examples with these patterns that are still difficult for static code analysis tools?

We define a difficult source code pattern as a vulnerability pattern that static analysis tools can not identify correctly. This means that either the tools

will report it as a vulnerability but it is not (false positive), or they do not report it as a vulnerability but it is (false negative). The term difficult does not represent a metric that includes different difficulties.

Section 13.2 points out related work. Background and methodology is described in section 13.3. The static code analysis scan results from the initial open source projects are described in section 13.4.1. The following section 13.4.2 shows the results of scanning the minimal working examples. Sections 13.5 and 13.6 describe the identified patterns in detail. To see if the patterns are difficult for static code analysis tools a verification is done in section 13.7. In section 13.8 we discuss why identifying such patterns is important and what static code analysis tools and developers can do to deal with such difficult patterns.

13.2 Related Work

Many research projects have evaluated static code analysis tools. Goseva-Popstojanova and Perhinschi [172] evaluated three commercial static code analysis tools. The Juliet database [177] and three open source projects related to Common Vulnerabilities and Exposures (CVE) reports were used to evaluate different permutations of security vulnerabilities. The results show that the tools had high false negative rates and none of the tools detected all vulnerabilities. Our approach also uses source code from open source projects related to CVE reports. Our objective is not to compare the tools with each other, but to instead find difficult source code patterns. Delaitre et al. [169] has similar results. They used as a data set source code from production software that they assumed had no vulnerabilities. Their results show that no tool detected all samples correctly. The data set is similar to ours with both complete open source projects related to CVE reports and specifically created samples. Díaz and Bermejo [170] also evaluated open source and commercial static code analysis tools. They used the test suite 45 and suite 46 from SAMATE [178] database as data sets. For the evaluation, they calculated and compared the F-measure [181] from each tool. AlBreiki and Mahmoud [167] evaluated three open source tools. They evaluated tools with different approaches. OWASP Yasca analyzed source code, FindBugs analyzed byte-code and Microsoft Code Analysis Tool .NET analyzed binary code. They used test cases based on top security issues from OWASP, CWE and SANS. Zhioua et al. [182] evaluated four static code analysis tools based on how they detect vulnerabilities and what techniques are used. They described how security issues and security properties are related to each other. Another approach from Khare et al. [173] evaluated

static code analysis tools on large samples with more than 10 million lines of code. The results showed that less than 10% of the vulnerabilities were reported.

Also, software fault injection has an effect on static code analysis tools [168]. Software fault injection actually affects the detection rate of the tested static code analysis tools. Primarily, it leads to false positive reports and existing vulnerabilities were not detected.

There is related work about identifying false positive source code patterns. Reynolds et al. [176] do a more similar approach as ours. They use extract the patterns by manual reviewing them. As test base they are using artificial vulnerabilities from the Juliet framework. Koc et al. [174] are using the previous data set to a classifier to identify problematic patterns. Overall the focus relies on identifying false positive reports. Our results are more focused on patterns that result in false negative reports and our data set is based on open source projects related to CVE reports.

13.3 Methodology

We use multiple static code analysis tools to find and evaluate problematic source code patterns. Problematic patterns result in false negative and false positive reports. This section provides background knowledge, explains the chosen commercial and open source static code analysis tools and how the problematic patterns are examined.

13.3.1 Background

A report from a static code analysis tool can either be true positive (TP), true negative (TN), false positive (FP) or false negative (FN). True positive and true negative reports mean the report is correct and it reported a vulnerability (positive) or it does not contain a vulnerability (negative), respectively. Problematic reports are false negative and false positive reports. A false negative report means that the tool did not report the vulnerability, but a vulnerability actually exists. To find difficult source code patterns that create false negative reports a data set is required that contains vulnerabilities. In contrast, a false positive report means that the tool reported a vulnerability which actually does not exist. To find patterns that create false positive reports a data set is required that does not contain a vulnerability.

13.3.2 Selected tools

It is not the goal to compare different static code analysis tools to each other. Instead, we want to find source code patterns that are difficult for static code analysis tools. First of all, three commercial static code analysis tools were used. Our licence agreement does not allow to publish the names of the tools. In this work the commercial tools are named Tool A, Tool B and Tool C. All of the tools are state-of-the-art that perform tainted data flow analysis. We focus on static code analysis and in case a tool provides more functionalities we only use the static analysis parts of the tools. Additionally, for the verification phase, two open source tools are used. We evaluated a collection of open source static code analysis tools [171], and selected Exakat (www.exakat.io) and Sonarcloud (https://sonarcloud.io), which are tools for finding security vulnerabilities in PHP and are still maintained (last update < 1 year ago). This approach does not review the internal details and the method that the tools are using in their analysis. As the commercial tools do not allow to examine how they work in detail, the open source tools are also seen as a black box. The goal is to find source code patterns that are difficult for static code analysis tools and reproduce them. On each pattern we describe in detail what problems the static code analysis tools have which prevented them from correctly analyzing the vulnerabilities.

13.3.3 Data set

The source code patterns should be as realistic as possible. We used a crawler to get source code of open source projects related to CVE reports (www.cve.mitre.org). It uses the categories from CVEDetails (www.cvedetails.com) to filter all CVE reports related to SQL Injection vulnerabilities. These reports are checked for having a confirmation link to a patch on GitHub (www.github.com). The patch itself is checked if it contains any PHP files. CVE reports from 2010 until 2016 were crawled. This ensures that the developers had enough time to patch the vulnerabilities and report the confirmation link to the CVE report. Additionally, all of the samples were manually reviewed to pin point the actually vulnerability. This also takes time and effort to ensure that the correct vulnerability was manually reviewed. Based on that filter criteria we randomly chose 50 CVE reports related to SQL Injection and PHP. We had to limit the number of CVEs considered for this work to be feasible by the researcher at this stage. Expanding the analysis beyond these 50 CVEs can be done in the future. The randomly chosen CVEs are shown in table 13.5 (appendix). For each CVE report, the developers provided a patch to fix the vulnerability. The source code

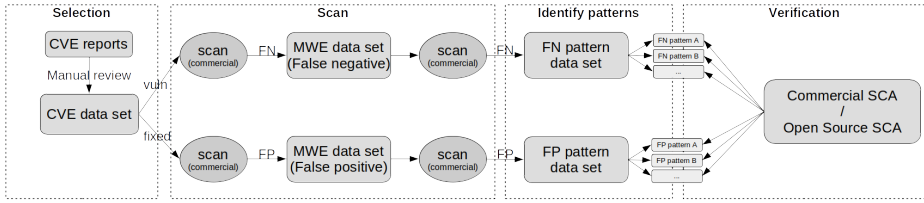


Figure 13.1: Vulnerability analysis process.

of the patch is used to create a data set that does not contain the reported vulnerability. This data set are used to find difficult source code patterns that potentially create false positive reports. In contrast, the revision of the source code samples before the patch were used to find source code patterns that might create false negative reports. Both data sets together are further called CVE data set. This data set, after being expanded with recent and more CVEs, can be used as a benchmark for studying and analysing difficult source code patterns.

13.3.4 Vulnerability analysis

Figure 13.1 shows the vulnerability analysis process. The process consists of 4 steps, which are (1) selection, (2) scanning, (3) identifying patterns, and (4) verification. As described previously, the CVE data set is split into a data set containing a reported SQL Injection and a data set that patched the vulnerability. All samples are scanned by the commercial static code analysis tools. If a tool does not find the vulnerability, a false negative is identified. In contrast, if a tool reports a vulnerability in the patched version, a false positive is identified. For each false negative and false positive reports, minimal working examples (MWE) were created as follows. First, a basic manual review of the initial source code from the CVE data set was done. This review process is simply tracking the data flow from the related source to the related sink. The identified data flow is then recreated to contain only the related source code. The related source code form the minimal working examples (cf. Figure 13.1). It can be noted that the goal of creating minimal working examples is to reduce the manual review effort, as its size is much smaller than the actual sample. This MWE data set is scanned by commercial tools again to check, if the problematic patterns are included or not. If a MWE sample is still creating a false negative or false positive reports, the minimal working example is reviewed to identify source code patterns. For each pattern, a sample is created. These samples are created

using a PHP file containing a simple SQL injection vulnerability. That file is modified to contain the source code pattern. If the pattern requires multiple files, additional files were added. This creates the next data set named FN pattern data set, which contains patterns that cause false negative reports, and FP pattern data set that cause false positive reports.

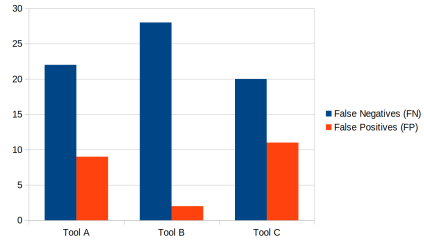
The final step in the analysis process is to verify which patterns are actually difficult. To do so, each of the FN/FP pattern data set entries are scanned again. This time the open source tools were used to check if these tools are performing similar to commercial tools. If the sample containing the pattern still creates a false positive or false negative report, a difficult source pattern is identified and confirmed.

13.4 Data set results

As we mentioned before, one of the contributions of the work is the creation of an initial data sets that can be used in the future as a benchmark for difficult source code patterns analysis. Although the data set is limited to 50 randomly chosen CVEs, but this work can be expanded in the future to include more comprehensive and actual material. In this section, we will explain the resulted CVE data sets as well as the minimal working example data set.

13.4.1 CVE data set results

In order to enable reproducibility of the results of this work, we list in table 13.5 50 vulnerable and 50 patched projects from the randomly chosen CVE. Figure 13.2a shows the false negatives and false positives that result from each tool. Even with that few samples (100) it shows a main problem of static code analysis tools. The main problem of static code analysis tools are finding the right balance between false negative and false positive rate. If you want to reduce the false positive rate, it will increase the false negative rate. Table 13.2b shows an overview of the results and the resulting static code analysis metrics. The accuracy is almost identical on all of the tools. The metrics also show that tool A and C are very similar in all aspects. Tool B is not as good in finding the vulnerability (recall), but it has a very low false alarm probability. Accordingly, if tool B reports a vulnerability, the chance is very high that a vulnerability actually exists. Nevertheless, it did not detect over 50% of the vulnerabilities.



(a) FN and FP statistic.

Metric	Tool A	Tool B	Tool C
Vulnerable projects	50	50	50
True Positive	28	22	30
False Negative	22	28	20
Patched projects	50	50	50
True Negative	41	48	39
False Positive	9	2	11
Accuracy	69%	70%	69%
Recall	65%	63%	66%
False Alarm Prob.	24%	8%	26%

(b) Metrics.

Figure 13.2: Results from the CVE data set.

13.4.2 Minimal working example data set

The minimal working examples were created based on the previous false negative and false positive reports. The minimal working data set can be found on GitHub [179]. Some of the initial data sets result in the same minimal working examples. This happened because some CVE entries were from the same open source project with different versions. In these samples the used source code was the same, so we only created one minimal working example. The minimal working example samples were scanned again to see if the important parts were found. Table 13.1 shows the results of the minimal working examples that should result in a false negative report from at least one tool. In two samples, all tools detected the vulnerability. Accordingly, we were not able to construct a minimal working example for these samples. Similar to the initial scans, Tool B has the most false negative reports. Tool C performs very well in this data set with only one false negative report. We reviewed the results manually to find the reason for the different results between the CVE data set and MWE data set. Tool C reports a SQL injection vulnerability if a string variable is concatenated that is used in a database sink. If no source is found, it still reports a potential SQL injection with a lower priority.

The results for the false positive data set is seen in table 13.2. Tool A and Tool C are reporting more false positives. The minimal working example (CVE-2011-4960) did not include the relevant source code patterns to create a false positive report because all tools correctly reported a true negative.

CVE related	Patterns	Tool A	Tool B	Tool C	Count of FN
CVE-2011-4960	ReflectionClass	FN	FN	✓	2
CVE-2012-0973	Get parameter function Singleton Func_get_args Function - sprintf	✓	FN	✓	1
CVE-2012-2762	-	✓	✓	✓	0
CVE-2012-3470	Inerhit query construction	✓	FN	✓	1
CVE-2012-3471	Eventmanager	✓	FN	✓	1
CVE-2012-5162 CVE-2013-3527 CVE-2015-4628 CVE-2016-9020 CVE-2016-9087 CVE-2016-9183 CVE-2016-7453 CVE-2016-9242 CVE-2016-9272 CVE-2016-9282	Get parameter function Database access object	FN	FN	✓	2
CVE-2013-2559	Database - static method Eventmanager	✓	FN	✓	1
CVE-2013-3081	Environment variable	✓	FN	✓	1
CVE-2013-3524	-	✓	✓	✓	0
CVE-2013-4789	Get parameter function	✓	FN	✓	1
CVE-2014-1608 CVE-2014-1609	SOAP	✓	FN	✓	1
CVE-2014-5017 CVE-2016-7400	Sub class get method Singleton - set Singleton - classes	✓	FN	✓	1
CVE-2014-9089	Sanitize only limited elements Explode - implode	✓	FN	✓	1
CVE-2014-9464	Singleton __get __set	✓	FN	✓	1
CVE-2014-9528 CVE-2014-9573	Get parameter function Complex query construction	✓	FN	✓	1
CVE-2015-4426	Stored class Json decode	FN	FN	✓	2
CVE-2016-2555	Database - global variable Sanitize function not initialized	FN	FN	✓	2
CVE-2016-5703	Database - global array Imported variable Imported sink	✓	FN	FN	2
Summary (FN)		4	16	1	

Table 13.1: Scan results from false negative MWE data set with 18 samples.

CVE related	Patterns	Tool A	Tool B	Tool C	Count of FP
CVE-2011-4802	Preg_match - check for number	FP	FP	FP	3
CVE-2011-4960	-	✓	✓	✓	0
CVE-2012-2762	Function - strreplace Sanitize if function exists	FP	✓	FP	2
CVE-2013-2559	Sanitize if function exists	FP	✓	FP	2
CVE-2013-4789	White listing	FP	✓	FP	2
CVE-2014-3773 CVE-2012-5162 CVE-2011-4341 CVE-2012-0973 CVE-2015-1471 CVE-2013-4879 CVE-2014-8351	Function - quote	FP	✓	FP	2
CVE-2015-2679	Function - htmlentities	FP	✓	FP	2
CVE-2016-7780	Sanitize function - global db variable	FP	✓	✓	1
Summary (FP)		7	1	6	

Table 13.2: Scan results from false positive MWE data set.

13.5 False negative source code patterns

The minimal working examples commonly contained multiple source code patterns. All the used source code patterns were reviewed and based on that pattern a sample was created containing the pattern only. The source code for each pattern can also be found on GitHub [179]. The patterns are categorized into source, concatenation, sink, sanitization and data flow. Figure 13.3 shows the categories and how they are found in a typical SQL injection vulnerability. The data flow (DF) patterns are in between other patterns. The different source code patterns are described in this section.

13.5.1 Sources

This section describes the source code patterns that are a source. All of the following patterns have some special parts that makes it difficult for static code analysis to detect it as a source:

- **Sub class get method**
  This source code pattern uses inheritance. The super class implements

Figure 13.3: Overview of source code pattern categories.

a method that uses the *method_ exist()* to check if the sub class implemented a get-method. If the getter method is implemented, it will be called using call-method-by-string. Call-method-by-string is an unconventional way of calling a method. Static code analysis tools have to parse the corresponding string that might also require additional data flow analysis. In our results only tool B was able to correctly parse the string to the corresponding method.

- **Get parameter function**
  A common found pattern in our data set was using wrapper methods for getting user data. It simply defines a method that uses common PHP (e.g. *$_ GET*) methods to get user data. This pattern includes a sanitization method which is not used by default. Tools have to check, if the sanitization method is enabled or not to correctly detect a vulnerability. The commercial tools were able to correctly detect a SQL vulnerability including this pattern.

- **SOAP**
  PHP allows to implement a Simple Object Access Protocol (SOAP). This pattern uses the *SOAPServer* class to implement a SOAP service. Parameters passed to that service are user data and potential dangerous. The problem for static code analysis tools relies on how such a service will be implemented in PHP. The *SOAPServer* class uses the *setClass* method to register a class name defined by a string. The tools have to backtrack the string value to actually know the class name of the provided SOAP service. If they have correctly backtracked the string value, the tools can mark all method parameters of the class as possible sources.

- **Import**
  A very simple source code pattern (Imported variable) uses an additional PHP file which is imported by the *require_ once* function. The imported file simply uses the *$_ GET* method to get user data and stores it in a PHP variable. This pattern requires that includes from other PHP files are parsed correctly. The commercial tools have no problem with includes from other PHP files.
  Another variant of that is the Stored class pattern. It is a bit more complex because the included file defines a class that stores tainted data in a class variable. Later on tainted data is retrieved by another get method that uses the class variable. Again the commercial tools have no problem tracking that tainted data is stored in class variables.

13.5.2 Insufficient sanitization

The false negative patterns are based on source code that contains a SQL Injection vulnerability. Accordingly, the user input was not sanitized correctly. We found the following two patterns that were containing insufficient sanitization methods:

- **Sanitize only limited elements**
  In this pattern user data is provided as an array. The data is iterated and sanitized by a white listing check. Only user data that is in the white list is allowed. The special part of this pattern is that only a limited amount of values are sanitized. In our data set only the first two elements of the array were sanitized. The tools have to check, if all elements in a array are checked. This requires the tool to keep track on how large an array might be. For example, if the array can only be the size of two elements, checking only the first two elements would be sufficient. This pattern sounds artificial, but our CVE data set actually had such a code pattern where only the first two elements were sanitized.

- **Sanitize function not initialized**
  This pattern uses a global string variable which defines the sanitization methods. This allows developers to decide which sanitization method will be used. A default method is already defined which only returns the provided string without any sanitization. The sanitization method is called using dynamic method invocation [175]. Again, this requires the static code analysis tools to parse the corresponding string to see what method is called. If the variable is defined global, it makes it even more difficult. It requires to correctly parse the PHP project. The global variable might be defined multiple times and the correct variable can only be tracked by parsing all the includes of different files. None of our tested tools were able to correctly detect this pattern.

13.5.3 Concatenation

Patterns in this category are describing how the SQL query is build. It describes the concatenation between the user input and the SQL query. We found following patterns:

- **Complex query construction**
  In this pattern the SQL query is constructed using multiple functions. It uses a class to store the query relevant data. The concatenation of

the SQL query is a concatenation of multiple method return values. Each of these method is a construction part of the query. For example, one method creates the *WHERE* clause of the query. The tainted data is stored in class variables. This requires the static code analysis tools to correctly track the tainted data in multiple method calls. Commonly, it includes also other functions from PHP. For example, the *implode* functions was found a lot of times to combine multiple parameters into one query. If only one part is incorrectly analyzed, the vulnerability is not detected.

- **Function - sprintf**
  This pattern uses the function *sprintf* to construct a SQL query in a c-like fashion. It allows to define a string with different specifiers which will be replaced by parameters. This is a common way to concatenate a string with variables. Static code analysis tools have to add the *sprintf* functions and correctly parse the variables to the correct positions in the string value. It also requires the tools to determine between a string replacement or just a fixed value replacement. For example, if only a number is inserted using the *sprintf* function, the tainted data is not tainted anymore. Accordingly, the *sprintf* function could also be used as a sanitization function. In our pattern, we used the string replacement which does not prevent any vulnerabilities.

13.5.4 Sink

Sinks are critical functions, if user input reaches it without any sanitization in between. Patterns of this category are different implementations for a SQL Injection sink.

- **Database access object**
  This was a very common source code pattern in our data set. A database access object (DAO) is a simple PHP class that stores the query relevant data as object variables. The DAO is able to construct the SQL query string using object methods. Similar to the the concatenation pattern (Complex query construction), the SQL query construction requires to track multiple method calls and tainted data stored in class variables. The main difference is that this pattern also stores the database object as a class variable. Calling corresponding functions on the DAO objects will construct the SQL query and also perform the query on the database. It then just returns the resulting data.
  A special case of this pattern is the (Inerhit query construction) pattern.

It also uses a database access object, but relevant implementations are defined by inherited classes. The tools have to know what sub class is used to decide, if a vulnerability exists or not.

- **Database object storage**
  In our CVE data set we found different ways of storing a database object. The connection to the database itself is usually only established once. Then the corresponding object is stored in different ways. It either is stored in a global array (Database - global array). This requires that the static code analysis tools are correctly tracking global arrays and what data is stored. Or it is stored in a global variable (Database - global variable). Another pattern was using static methods to connect to an database and to statically get the corresponding database object (Database - static method).

13.5.5 Data flow

The data flow is a relevant part of SQL Injection vulnerabilities. The data passes different source code pattern between the source and sink. These patterns are based on SQL Injection vulnerabilities, but data flow source code patterns are also relevant for other vulnerability types.

- **Eventmanager**
  An Eventmanager is used to create a system based on events. The implementation uses the static class methods *add* and *run*. The method *add* allows to add callbacks to specific events. If the *run* method is used, a specific event is run and all related callbacks are called. The callbacks are stored in static class variables. Static code analysis tools have to track all callbacks that are stored in the Eventmanager class. Programs written using an Eventmanager are completely different than a objective oriented programming style. It is much more difficult to parse all the possibilities of what kind of events occur and it might even be unpredictable. Our pattern sample uses predefined events that always end up in a vulnerability.

- **PHP pass through functions**
  There are multiple PHP functions that returns data from a parameter (pass through). Static code analysis tools have to define these functions as passing through tainted data. In our data set we found following functions *explode*, *implode* and *json_ decode*. None of these functions are changing the data in a way that it prevents a SQL injection vulnerability.

Accordingly, the tools should define these function as pass through functions.

- **Dynamic PHP functionality**
  PHP provides a lot of language features. It has functionality that allows to dynamically call functions and methods. Our data set showed the usage of *func_get_args* function. This function allows to get function parameters without defining them at the function definition. This makes it very difficult for static code analysis tools to correctly track the parameters. The parameters are returned as an array.
  Also our data set showed the definition of *__get* and *__set* class methods. These methods are called when a class variable is accessed that is not defined in the class definition. In the set method the class variable name and the value are passed as parameters. The get method only has the class variable name as parameter. Our implementation just stores the value for the corresponding class variable name and returns the corresponding value on the get method. Nevertheless, the implementation might differ and static code analysis tools have to analyze the methods. If the methods are analyzed then all corresponding class variable accesses without a class variable definition have to be tracked to correctly analyze the program. Our pattern sample is very simple to see if static code analysis tools are analyzing the dynamic get and set methods.
  Another dynamic feature of PHP is setting and getting environment variables (Environment variable). It provides the function *putenv* to set an environment variable. The parameter type is string. The string itself requires to be in a specific format (*"Varname=Value"*) to actually set an environment variable. The corresponding function *getenv* is used to get the value of an environment variable. The parameter of the function is a string that defines the variable name. Static code analysis tools have to analyze and backtrack the corresponding strings to correctly analyze this pattern.

- **Plugin support**
  This a very complex source code pattern. It actually might be more a architecture, but it is commonly found in our CVE data set. It uses a plugin structure that allows to easily add more modules. In our sample for each plugin a controller and view class has to be implemented. These implementations are sub classes from template classes. These implementations have to be in a subdirectory in a fixed plugin structure. The source code pattern parses the plugin folder for valid

plugin implementations. A valid plugin implementation can then be accessed as it would be normal PHP web page. For this access a router class is implemented which routes to the correct plugin. The usage of plugins as a developer is convenient. It allows to split the programs in different modules. In contrast, static code analysis tools have problems analyzing plugin supported programs. First of all the programs has to be analyzed to see that the program itself has plugin support. The analyze also has to find out what files are included from a plugin. Because plugin support is usually not fixed to a specific number of plugins, the loading off such plugins is dynamic. Loaded plugins are stored in class variables and corresponding PHP files are loaded.

- **Singleton**
  The simple singleton sample is just implementing the common known singleton pattern. Singleton is a source code pattern that allows to access only one instance of an object. The singleton itself has a get method to access the *$_GET* parameter. Because a singleton can be accessed from everywhere, the static code analysis tools have to track all possible ways of calling the singleton. Our sample is just a procedural calling of the singleton.
  Our CVE data set also showed that the singleton pattern was used with different implementations. The Singleton - classes pattern allows to get class objects by a name. The name itself is a string variable that requires the static code analysis tools to actually analyze the string variable to know what class object is returned.
  Another pattern (Singleton - set) returns one instance of an object. The object itself is not predefined. Initial a corresponding set method has to be used to set the singleton object. Afterwards the singleton object can be accessed from everywhere.
  The ReflectionClass pattern is not a singleton per definition. It uses a static class implementation to create new class objects. Accordingly, you can access the class from everywhere, but you will always get a new object. It uses a combination of the PHP functions *func_get_args* and *array_shift* to get the class name and parameters provided as a function parameter. The new instance of the class is created using the *ReflectionClass* from PHP standard library. The class name itself is again a string parameter that has to be analyzed by the static code analysis tools to get the corresponding class of the returned object.

13.6 False positive source code patterns

Source code patterns in this category are patterns that developers used to fix the reported vulnerabilities. The described source code patterns are sufficient to prevent SQL injections, but still static code analysis tools are reporting a vulnerability.

- **Official sanitization**
  Database driver usually provide a sanitization function to sanitize tainted data. We used the *quote* function provided by the PDO class from PHP. The initial CVE data set also contained old source code samples that used functions like *mysql_ real_ escape_ string*. These functions are not supported anymore and the PHP documentation provides the *quote* as an alternative. Accordingly, we added the *quotes* function as pattern. This function should be added to a static code analysis tool as a sanitization method. As described in next section, this sanitization method is correctly detected by all the tested static code analysis tools.
  Another pattern uses the *htmlentities* function. This is not a specific function to prevent SQL injection vulnerabilities. This source code pattern just uses the *htmlentities* function to sanitize the user data. But it requires that the SQL query itself adds quotes around the sanitized data to ensure that a SQL injection is not possible. Accordingly, static code analysis tools have to analyze the query to see, if the sanitization with *htmlentities* is sufficient or not. This is a problem of many sanitization method, that they are sufficient enough in a specific context. Some of them are sufficient enough for a specific vulnerability type and some of the sanitization methods like the *htmlentities* is only sufficient enough based on the SQL query statement.

- **Custom Sanitization**
  The CVE data set revealed multiple custom sanitization implementations. The function *preg_ match* can be used to check a string value based on a regular expression (regex). The regular expression is an important part because based on that expression a SQL injection can be prevented or still might be insufficient. Static code analysis tools have to analyze the regular expression to see, if the sanitization method is sufficient. This pattern uses a regular expression that checks, if the string value only contains numbers. Accordingly, this is sufficient to prevent any SQL injection attacks.
  Another pattern we found, uses the *str_ replace* function to replace

any dangerous characters. The initial sample replaced all apostrophes and the SQL query itself puts the user data inside apostrophes. This time the static code analysis tools have to the analyze of the regular expression and the SQL statement to determine, if the sanitization method is sufficient or not.

- **White listing**
  White listing is a common way to mitigate any attack. Only fixed inputs are allowed. These fixed inputs should be chosen that they are not creating any attack possibilities. The implementation of white listing can differ. Our implementation based on the CVE data set an array is defined that contains all the allowed inputs. The *isset* function is used to check if the user data is contained in the white list array. Accordingly, static code analysis tools have to analyze the content of the white list array to see what inputs are possible. The creation of the array might be complex. A static code analysis tools also has to check all the possible inputs, if any of these inputs might still result in a vulnerability.

- **Dynamic defined sanitization functions**
  Many of the CVE data set source code samples are from frameworks. They allow to define what kind of database is used. Also some of them allow to define what sanitization method will be used. The Sanitize function - global db variable pattern uses a wrapper method for the *escapeString* function. The wrapper function is implemented as a static class function. The implementation itself uses a global defined variable for accessing the database connection object. This object provides the relevant sanitization method (*escapeString*). This global variable is defined in the initialization process. This makes it very difficult for static code analysis tools because they have to analyze what sanitization function is defined in the initialization process. Based on what function is used, it also may require to analyze the SQL query to see, if it is sufficient.
  The Sanitize if function exists pattern uses a wrapper function for the *quote* function. The wrapper function uses the *function_ exists* function to check, if the sanitize function actually exists. In our sample the function exists because it checks for a standard php library. Nevertheless, if an older PHP version is used, it might not exist. This makes the pattern also PHP version dependent. The static code analysis tools also have to know what PHP version is used.

13.7 Verification

The difficult source code patterns were scanned again to verify what patterns are actually difficult. A simple sample was used which contains a simple SQL injection vulnerability. For each of the previous described patterns the simple sample was modified to contain the pattern. The evaluation also used the open source static code analysis tools. The created source code patterns can be found on GitHub [179].

Table 13.3 shows the result for the false negative patterns. Two identified source code patterns were not creating a false negative report. Accordingly, these two patterns (Imported sink, Database - wrapper) are not difficult for state of the art static code analysis tools. The open source tools already have problem with simple patterns like Get parameter function, Imported variable, Stored class and Singleton. The different singleton implementations makes it difficult even for the commercial tools. Interestingly, the open source tools detected some of the singleton implementations correctly. The ReflectionClass pattern was difficult for all of the tested tools. That pattern includes different dynamic PHP features and the combination of creating a class object based on a string value. Accordingly, it includes many already difficult sub patterns. Also the Plugin supportis a pattern that includes different PHP features together to create a plugin support. None of the tools were able to detect them.

Table 13.4 shows the result for source code patterns related to false positive reports. The Function - quote pattern was not difficult for any of the tested tools. As already state the initial CVE data set also contained old source code using outdated sanitization methods. These methods were actually creating false positive reports because the static code analysis tools did not have them in their list of sanitization functions. It also shows that all of the tested static code analysis tools are checking for sanitization methods. The results show that the tools are very different on detecting sanitization approaches. Some of them are even considering custom sanitization attempts and other tools just ignore them.

False negative pattern	Tool A	Tool B	Tool C	Exakat	Sonarcloud	Count of FN
Source						
Sub class get method	FN	✓	FN	FN	FN	4
Get parameter function	✓	✓	✓	FN	FN	2
SOAP	✓	FN	FN	FN	FN	4
Imported variable	✓	✓	✓	FN	FN	2
Stored class	✓	✓	✓	FN	FN	2
Insufficient sanitization						
Sanitize only limited elements	✓	FN	✓	FN	✓	2
Sanitize function not initialized	FN	FN	FN	FN	FN	5
Concatenation						
Complex query construction	✓	FN	✓	FN	✓	2
Function - sprintf	✓	FN	✓	FN	✓	2
Sink						
Database access object	FN	FN	✓	FN	✓	3
Database - global array	✓	✓	✓	✓	FN	1
Database - global variable	✓	✓	✓	✓	FN	1
Database - static method	✓	FN	✓	FN	FN	3
Inerhit query construction	✓	FN	✓	FN	FN	3
Database - wrapper	✓	✓	✓	✓	✓	0
Imported sink	✓	✓	✓	✓	✓	0
Data flow (DF)						
Eventmanager	FN	FN	FN	✓	FN	4
Explode - implode	✓	FN	✓	FN	✓	2
Func_get_args	✓	FN	FN	FN	FN	4
__get __set	✓	✓	FN	FN	✓	2
Json decode	FN	FN	✓	FN	FN	4
Plugin support	✓	FN	FN	FN	FN	4
Environment variable	FN	FN	✓	✓	FN	3
Singleton	✓	✓	✓	FN	FN	2
Singleton - classes	✓	FN	FN	✓	FN	3
Singleton - set	FN	✓	✓	✓	FN	2
ReflectionClass	FN	FN	FN	FN	FN	5
Summary (FN)	7	15	9	19	18	

Table 13.3: Scan results for different false negative patterns.

False positive pattern	Tool A	Tool B	Tool C	Exakat	Sonarcloud	Count of FP
Sanitization						
Preg_match - check for number	✓	FP	FP	FP	FP	4
Function - quote	✓	✓	✓	✓	✓	0
Sanitize if function exists	FP	✓	FP	✓	FP	3
Sanitize function - global db variable	FP	✓	✓	✓	✓	1
Function - htmlentities	FP	✓	FP	✓	✓	2
Function - strreplace	FP	✓	FP	✓	✓	2
White listing	✓	FP	FP	FP	FP	4
Summary (FP)	4	2	5	2	3	

Table 13.4: Scan results for different false positive patterns.

13.8 Discussion

The evaluation shows that almost all of the identified patterns are difficult for at least one static code analysis tool. Some of these patterns are just common programming functionality provided by PHP. Such functionality should not be difficult for modern static code analysis tools. Most patterns are related to SQL injection vulnerabilities, except the patterns in the data flow category. These are patterns transferring user data from one point to another. These patterns are interfering the data flow algorithm from the static code analysis tool. Accordingly, these patterns are not difficult just for SQL injection vulnerabilities, instead they are difficult for all vulnerability types that require user input reaching critical functions. The developers of the static code analysis tools should be able to improve their algorithms to get fewer false negative and false positive reports. Nevertheless, some of the patterns are not that easy to be detected correctly, especially if the patterns contain dynamic language features of PHP. If developers use such a feature they should only use it if it is necessary. As our results show as more of such dynamic features are included, the more false negative reports occur. Also if regular expressions are involved, the tools have to parse the expression. Based on the expression, the sanitization might be sufficient or not. Usually the expression is also related to the SQL query. Our presented patterns can be prevented in the development phase. These patterns can be replaced by code patterns that can be easily detected by static code analysis tools. For example, using the *str_replace* function for replacing critical characters can be replaced by using common known sanitization methods provided by the database library. Static code analysis tools know that these sanitization methods are sufficient to prevent any SQL injection attacks. Accordingly, the tool is then not reporting a false negative report.

This work required different manual review steps. The source code of the

different open source projects were initially scanned by the tools. The results had to be reviewed manually to find all the false negative and false positive reports. Because of the manual review process the data set was limited. 50 false negative samples and 50 false positive samples were used to find the previously described difficult source code patterns. Some of the samples were even from the same open source project. Because of the small data set, we can be sure that there are still more difficult source code patterns for static code analysis tools. The different tools also could not be compared to each other because of the small data set. The results show a tendency that commercial tools outperform open source tools. Especially that the difficult source code patterns were identified specifically based on the false negative/false positive reports from the commercial tools. There were a lot more difficult patterns for false negative reports found than for false positive results. The reason for this is that we only reviewed reports based on the patched versions. If a tool did not report the vulnerability in the vulnerable version, the modifications of the patch will not create a false positive report. Finding a solution for the manual review steps would allow to research for difficult source code patterns on a broad scale.

The CVE data set itself does only include reports until end of 2016. The reason is that the manual reviewing of the source code pattern takes a significant amount of time. The resulting patterns are all updated to the up to date PHP version with corresponding functions. The tested static code analysis tools are all state of the art and the pattern are still difficult for them. A newer CVE data set might introduce even more difficult source code patterns. Nevertheless, our results show that the patterns we created are still difficult.

13.9 Conclusion

The goal to find difficult source code patterns was successfully achieved. The review of 50 open source projects containing vulnerable and patched versions revealed 25 difficult source code patterns for false negative reports and 6 difficult source code patterns for false positive reports. The verification shows that modifying simple vulnerabilities with these patterns are still difficult for static code analysis tools. The dynamic language features of PHP are nice for programmers, but for static code analysis tools they are very difficult. The results show that most identified patterns are data flow patterns. Many patterns should be detected by modern static code analysis tools and their developers should improve the algorithms based on our results. Nevertheless, some patterns can also already be mitigated during the development phase.

Developers should know what patterns are difficult for static code analysis tools. Our patterns can be used as learning examples for teaching higher level of software security.

References

[167] H. H. AlBreiki and Q. H. Mahmoud. Evaluation of static analysis tools for software security. *2014 10th International Conference on Innovations in Information Technology (IIT)*, pages 93–98, 2014.

[168] T. Basso, P. C. S. Fernandes, M. Jino, and R. Moraes. Analysis of the effect of Java software faults on security vulnerabilities and their detection by commercial web vulnerability scanner tool. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 150–155, 2010.

[169] A. Delaitre, B. Stivalet, E. Fong, and V. Okun. Evaluating Bug Finders – Test and Measurement of Static Code Analyzers. *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)*, pages 14–20, 2015.

[170] G. Díaz and J. R. Bermejo. Static analysis of source code security: Assessment of tools against SAMATE tests. *Information and Software Technology*, 55(8):1462–1476, 2013.

[171] Exakat. PHP Static code analysis tools list. `https://github.com/exakat/php-static-analysis-tools`, 2019.

[172] K. Goseva-Popstojanova and A. Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, 2015. ISSN 09505849.

[173] S. Khare, S. Saraswat, and S. Kumar. Static Program Analysis of Large Embedded Code Base: An Experience. *Proceedings of the 4th India Software Engineering Conference 2011*, pages 99–102, 2011.

[174] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, page 35–42, 2017.

[175] PHP. `https://www.php.net/manual/language.namespaces.dynamic.php`, 2019.

[176] Z. P. Reynolds, A. B. Jayanth, U. Koc, A. A. Porter, R. R. Raje, and J. H. Hill. Identifying and documenting false positive patterns generated by static code analysis tools. In *2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER IP)*, pages 55–61, 2017.

[177] Samate. Juliet Test Suite. `http://samate.nist.gov/SRD/testsuite.php`, 2019.

[178] Samate. SARD. `https://samate.nist.gov/SARD/testsuite.php`, 2019.

[179] F. Schuckert. Patterns. `https://github.com/fschuckert/sca\_patterns`, 2019.

[180] F. Schuckert, B. Katt, and H. Langweg. Difficult XSS Code Patterns for Static Code Analysis Tools. *1st Model-Driven Simulation and Training Environment for Cybersecurity*, 2019.

[181] C. J. Van Rijsbergen. *Information Retrieval, 2nd edition.* 1979.

[182] Z. Zhioua, S. Short, and Y. Roudier. Static Code Analysis for Software Security Verification: Problems and Approaches. In *Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International*, pages 102–109, 2014.

Appendix

Table 13.5: CVE data set.

CVE	Github link	FN (A)	FN (B)	FN (C)	FP (A)	FP (B)	FP (C)
CVE-2016-9283	https://github.com/exponentcms/exponent-cms/commit/559792be7274e731bfcb3935f5beec7749e9ce9						
CVE-2016-9282	https://github.com/exponentcms/exponent-cms/commit/e83721a5b9fcc88e1141a8fb29c3d1bd522257c1	x	x	x			
CVE-2016-9272	https://github.com/exponentcms/exponent-cms/commit/ffb2038de4c603931b785a4c3ec69cfd06181ba	x	x	x			
CVE-2016-9242	https://github.com/exponentcms/exponent-cms/commit/6172f67620ac13fc2f4e9d650c61937d48e9ecb9	x	x	x			
CVE-2016-9183	https://github.com/exponentcms/exponent-cms/commit/3b3557e9f6ba193a4c23c8ce5498fa285dddf3f3	x	x	x			
CVE-2016-9134	https://github.com/exponentcms/exponent-cms/commit/45a7a62797e64e8abbac35d4859097c26f1874b1						
CVE-2016-9087	https://github.com/exponentcms/exponent-cms/commit/fdafb5ec97838e4edbd685f587f28d3174ebb3db	x	x	x			
CVE-2016-9020	https://github.com/exponentcms/exponent-cms/commit/fdafb5ec97838e4edbd685f587f28d3174ebb3db	x	x	x			
CVE-2016-7788	https://github.com/exponentcms/exponent-cms/commit/fdafb5ec97838e4edbd685f587f28d3174ebb3db						
CVE-2016-7781	https://github.com/exponentcms/exponent-cms/commit/fdafb5ec97838e4edbd685f587f28d3174ebb3db						
CVE-2016-7780	https://github.com/exponentcms/exponent-cms/commit/a8efd9ca71fc9b8b843ad0910d435d237482ee31				x		
CVE-2016-7453	https://github.com/exponentcms/exponent-cms/commit/c1092f167cc6c78dc8bf9bf149946c5219413df3	x	x				
CVE-2016-7405	https://github.com/ADOdb/ADOdb/commit/bd9eca9f40220f9918ec3cc7ae9ef422b3e448b8						
CVE-2016-7400	https://github.com/exponentcms/exponent-cms/commit/e916702a91a6342bbab483a2be2ba2f1dca3aa3		x				
CVE-2016-5703	https://github.com/phpmyadmin/phpmyadmin/commit/ef6c66dca1b0cb0a1a482477938cfc859d2baee3	x		x			
CVE-2016-2555	https://github.com/atutor/ATutor/commit/945a9dca01def8536516088da30fe6a4b7e9fa85	x	x	x			
CVE-2015-5078	https://github.com/LimeSurvey/LimeSurvey/commit/65d717415a271242b9a30a5330d4eabac1c1a837						
CVE-2015-4628	https://github.com/LimeSurvey/LimeSurvey/commit/b09edc0dbd18d8459ade4c7c941e562c16564f9e	x	x	x			
CVE-2015-4426	https://github.com/pimcore/pimcore/commit/1c6692e8287deed7f3356b6a1e2e9b7fe4e858dd	x	x	x			
CVE-2015-2679	https://github.com/semplon/GeniXCMS/commit/6982454883433961851b49e7482ee5b25541815				x	x	x
CVE-2015-1471	https://github.com/delta/pragyan/commit/c93bc100ec93fc78940fbdca9b6b009101858309				x		x
CVE-2014-9573	https://github.com/mantisbt/mantisbt/commit/69c2d28d	x	x				
CVE-2014-9528	https://github.com/humhub/humhub/commit/febb89ab823d0bd16246c6cf460addabb6d7a01d4	x	x	x			
CVE-2014-9464	https://github.com/microweber/microweber/commit/4ee09f9dda35cd1b15daa351f335c2a4a0538d29	x	x	x			
CVE-2014-9096	https://github.com/Pligg/pligg-cms/commit/efb967b944375cd3ea3cd84c80d86d339dbe030e						
CVE-2014-9089	https://github.com/mantisbt/mantisbt/commit/b0021673ab23249244119bde3c7fcecd4daa4e7f		x				
CVE-2014-8351	https://github.com/LaboCNIL/CookieViz/commit/489b6050f6c53fe7b24c4bed3eeb9c25543960e2						x
CVE-2014-5017	https://github.com/LimeSurvey/LimeSurvey/commit/9938bcd1df8ea27052557c722a67b00c0e7d6cb6	x	x	x			
CVE-2014-3773	https://github.com/nilsteampassnet/TeamPass/commit/7715512f2bd5659cc69e063a1c513c19e384340f				x		x
CVE-2014-1609	https://github.com/mantisbt/mantisbt/commit/7efc0175f0853e18ebfacedfd2374c4179028b3f	x	x				
CVE-2014-1608	https://github.com/mantisbt/mantisbt/commit/00b4c17088fa56594d85fe46b6c6057bb3421102		x				
CVE-2014-1401	https://github.com/auracms/AuraCMS/commit/790f66ffbc4f23a6e13636fc79d0aa1a7d81e747						
CVE-2014-10033	https://github.com/gburton/oscommerce2/commit/e4d90eccd7d9072ebe78da4c38fb048bfc31c902						
CVE-2013-4879	https://github.com/bigtreecms/BigTree-CMS/commit/c5f27bf66a7f35bd3daeb5f693f3e2493f51b1f3						x
CVE-2013-4789	https://github.com/Cotonti/Cotonti/commit/45eec046391afabb676b62b9201da0cd5303604b4	x	x				x
CVE-2013-3527	https://github.com/vanillaforums/Garden/commit/830785911bc4d263e77d2a2ca283100997755290d	x	x	x			
CVE-2013-3524	https://github.com/DavidJClark/phpVMS-PopUpNews/commit/efaffa04ef87db1722d69ac7bfc07be71ce2dccf	x	x	x			
CVE-2013-3081	https://github.com/JojoCMS/Jojo-CMS/commit/972757c4500d94b4b1306bf092e678add3a987d8	x	x	x			
CVE-2013-2559	https://github.com/symphonycms/symphony-2/commit/6c8aa4e9c810994f763283748742686 7ce50f468		x		x		x
CVE-2012-5162	https://github.com/osclass/OSClass/commit/ff7ef8a97301aaaf6a97fe46c2c27981a86b4e2f		x	x	x		
CVE-2012-3471	https://github.com/ushahidi/Ushahidi_Web/commit/3f14fa0		x	x			
CVE-2012-3470	https://github.com/ushahidi/Ushahidi_Web/commit/3301e48		x	x			
CVE-2012-3469	https://github.com/ushahidi/Ushahidi_Web/commit/e0e2b66						
CVE-2012-3468	https://github.com/ushahidi/Ushahidi_Web/commit/fdb48d1						
CVE-2012-2762	https://github.com/s9y/Serendipity/commit/87153991d06bc18fe4af05f97810487c4a340a92	x				x	x
CVE-2012-0973	https://github.com/osclass/OSClass/commit/ff7ef8a97301aaaf6a97fe46c2c27981a86b4e2f	x	x	x	x		
CVE-2011-4960	https://github.com/silverstripe/sapphire/commit/fef7c32		x				x
CVE-2011-4959	https://github.com/silverstripe/sapphire/commit/73cca09						
CVE-2011-4802	https://github.com/Dolibarr/dolibarr/commit/c539155d6ac2f5b6ea75b87a16f298c0090e535a				x		x
CVE-2011-4341	https://github.com/symphonycms/symphony-2/commit/476e4926e2773588eab10dd3036f27e1411521b5				x		x
		22	28	20	9	2	11

Chapter 14

Insecurity Refactoring: Automated Injection of Vulnerabilities in Source Code

14.1 Introduction

Automating the injection of vulnerabilities into a codebase can yield valuable knowledge for two cases: How easy is it for an attacker to quickly add vulnerabilities in a short period of time? This is a scenario that could be observed in attacks on PHP Git repositories where a backdoor was inserted [202]. A second case is training of software developers for inspections by the help of complex code samples. Oftentimes, training samples containing vulnerabilities are manually created [190]. This requires a significant effort and usually leads to small applications built around few vulnerabilities. Software inspections in the field, however, deals with large and complex applications where vulnerabilities are not easy to spot. Using existing applications with known vulnerabilities is insufficient for training situations, because learners are able to find the vulnerabilities documented in publicly available databases. Hence, automatically generated vulnerabilities have been proposed, e.g., by [209] and [185]. Those automatically generated samples are artificial and can be used to benchmark tools for static code analysis.

Our approach is to automatically create learning examples by modifying existing large projects. To achieve that, we use vulnerability patterns to inject vulnerabilities into open source projects. The use of existing projects ensures that the context of a vulnerability is as real as possible. We created source code patterns by examining vulnerabilities and corresponding fixes in

source code spanning a period of multiple years [205, 207]. The source code originated from real applications for which vulnerabilities had been reported with an assigned CVE-ID (CVE: Common Vulnerabilities and Exposures). Our contribution answers the following questions:

- Are static code analysis and refactoring valid approaches to inject vulnerabilities in existing projects?
- How can source code patterns from recent vulnerabilities be represented within the injected vulnerabilities?
- Are the insecurity refactored applications useful for teaching software security?

Insecurity Refactoring is a change to the internal structure of software to inject a vulnerability without changing the observable behavior in a normal use case scenario. We proposed a technique to conduct insecurity refactoring using static code analysis methods. An Adversary Controlled Input Dataflow (ACID) tree is constructed to find possible injection paths. These possible injection paths are then transformed into vulnerabilities using patterns from known vulnerabilities. The implementation is evaluated on open source projects to find possible injection paths and inject vulnerabilities.

Section 14.2 provides an overview of work related to Insecurity Refactoring and describes the definition of the Code Property Graph. Section 14.3 defines and formulates the methods and concepts proposed in our methodology. Section 14.4 formulates the tree construction mechanism that is used for the Insecurity Refactoring process. The definitions of Insecurity Refactoring are described in section 14.5. Section 14.6 describes the source code pattern language PL/V and how vulnerabilities are injected. Followed by section 14.7, in which the approach is evaluated on open source projects from GitHub. In this context the usefulness of the methodology in software security training is evaluated by an experiment involved two groups with different skill levels. The final section 14.8 points out problems and concerns of Insecurity Refactoring.

14.2 Background

Thomas et al. [210] developed a tool that replaces database queries with prepared statements to remove potential SQL Injection vulnerabilities. They present a prepared statement replacement algorithm (PSR-Algorithm) that separates the SQL query string from any input strings. They evaluated their tool on IT security training projects like WebGoat and 94% of their refactored prepared statements prevented SQL Injection attacks. Maruyama and Omori

[197] present a security-aware refactoring tool. Normal refactoring approaches can create unintended security issues. The focus relies on accessibility of class variables. Refactoring approaches can change the accessibility without changing the external behavior that might result in a security issue. Their tool actually checks for such security issues and provides refactoring approaches that do not create such issues.

Dolan-Gavitt et al. [189] created a tool named Large-scale Automated Vulnerability Addition (LAVA) that uses dynamic taint analysis to find locations to inject vulnerabilities in C/C++ projects. The dynamic approach makes the data flow analysis easier. The injected vulnerabilities themselves are artificial. Nevertheless, injecting vulnerabilities in real projects provides a more realistic scenario than manually creating a small project that contains a vulnerability. Also a LAVA-M data set was released that contains many injected vulnerabilities in C-projects. That data set is commonly used to evaluate modern fuzzers [195] [203].

Pewny and Holz [200] developed the EvilCoder tool similar to LAVA that injects bugs in C-projects. Similar to this work, the Code Property Graph is used to find potential injection locations. The focus relies on memory critical functions and corresponding security checks. These security checks are replaced by insufficient checks to create vulnerabilities. An evaluation on four open source projects shows the potential to conduct injections at many different locations. However, they cannot ensure that an injected vulnerability is exploitable.

Our approach uses the Code Property Graph defined by Yamaguchi et al. [211]. The Code Property Graph combines an Abstract Syntax Tree, Control Flow Graph and Program Dependence Graph into a single graph. They use the graph to find vulnerabilities in C/C++ projects. Backes et al. [184] extended the Code Property Graph to support PHP. We use this PHP graph to create the Adversary Controlled Input Dataflow tree. Alhuzali et al. [183] also used the Code Property Graph to find vulnerabilities in PHP projects. They added support to automatically create exploits for the discovered vulnerabilities. The results show that the Code Property Graph is very useful to discover vulnerabilities in C/C++ and PHP.

The usage of insecurity refactoring to create learning example seems promising. Schreuders et al. [204] developed the Security Scenario Generator (SecGen) that allows to create multiple virtual machines containing different vulnerabilities. The vulnerabilities are defined by modules. Based on the module description, vulnerabilities can be nested and hints can be placed. A survey that has been used to evaluate the usage of such generated virtual machines is helpful as learning examples. Yamin and Katt [213] [212] developed

a similar framework to automatically create full cyber security ranges that setup multiple virtual machines. The focus relies on creating a large number of virtual machines and the cyber security ranges are used for different scenarios like attacker/defence CTF events. Based on the scenario different vulnerabilities are injected to the machines via ssh. For example, injected vulnerabilities can be weak passwords, misconfigurations, components with known vulnerabilities, etc. Chapman et al. [187] designed the PicoCTF tool and hosted a capture the flag (CTF) event where approximately 2,000 teams participated. The approach was game based. The tasks can either be viewed in computer game style including a story or in a classical text view. A survey has been used to evaluate the approach. The results show that the approach is useful and many other CTF events have used the PicoCTF tool. Burket et al. [186] explain the automatic problem generation (APG) for PicoCTF. The APG allows to generate CTF tasks that differ for each attending team. A templated autogen problem uses a fixed template and multiple inputs (e.g. flag) to generate the CTF task. This allows to detect key sharing between teams but does not prevent sharing the *method* to solve the task between teams. In contrast, challenges that are automatically generated without a fixed template have problems with consistent difficulties, bug prevention, scalability and deployment. Another PicoCTF event has been held using a templated autogen to reveal that key-sharing actually exists and can be detected by the approach.

14.2.1 Code Property Graph

This section explains the definitions introduced by Yamaguchi et al. [211] of the Code Property Graph (CPG) and traversal functions.

Definition 1. A Code Property Graph $G = (V, E, \lambda, \mu)$ is a directed, edge-labeled and attributed multigraph. V is the set of nodes, $E \subseteq (V \times V)$ is the set of directed edges. The labels of these edges are defined by $\lambda : E \to \Sigma$ where alphabet Σ represents all edge names. Properties for edges and nodes are assigned by $\mu : (V \cup E) \times K \to S$. K is a set of property keys and S is the set of property values.

The Code Property Graph is based on the Abstract Syntax Tree (AST). The Abstract Syntax Tree is defined as follows:

$$G_A = (V_A, E_A, \lambda_A, \mu_A) \tag{14.1}$$

The Abstract Syntax Tree has one kind of edge labels (*parent_of*), which is defined in the set λ_A. The set μ_A contains property assignments for every

node of the Abstract Syntax Tree. For example, the name of a variable is stored as a property and the value is the variable name.

The Control Flow Graph is defined as follows:

$$G_C = (V_C, E_C, \lambda_C, \varnothing) \tag{14.2}$$

The nodes $V_C \subseteq V_A$ are statements of the programming language. For example, an assignment is a statement. Edges E_C represent the possible control flow from a statement to another statement. For the edges, only one kind of label (*flows_ to*) exists that is defined in λ_C. No properties are stored for the Control Flow Graph.

The Program Dependence Graph defines where variables are used and resolves function calls. The definition for the Program Dependence Graph is as follows:

$$G_P = (V_P, E_P, \lambda_P, \mu_P) \tag{14.3}$$

The nodes $V_P \subseteq V_A$ are the same nodes as of the Abstract Syntax Tree. Edges E_P either represent function calls or variable usage. Function calls have the edge label *calls*. The variable usages have the label *reaches* that point from the variable definitions to the statements where the variables are used. These edge labels are defined in λ_P. For the *reaches* edges the property μ_P defines the variable name of the variable definition.

Definition 2. A traversal is defined as a function $\tau : \mathbb{P}(V) \to \mathbb{P}(V)$ that maps a set of nodes to another set of nodes according to a Code Property Graph G, where $\mathbb{P}(V)$ is the power set of V.

The following function definition allows to iterate over an edge:

$$OUT_l(X) = \bigcup_{v \in X} \{u : (v,u) \in E \wedge \lambda((v,u)) = l\} \tag{14.4}$$

$$OUT_l^{k,s}(X) = \bigcup_{v \in X} \{u : (v,u) \in E \wedge \lambda((v,u)) = l \wedge \mu((v,u),k) = s\} \tag{14.5}$$

$$IN_l(X) = \bigcup_{u \in X} \{v : (v,u) \in E \wedge \lambda((v,u)) = l\} \tag{14.6}$$

$$IN_l^{k,s}(X) = \bigcup_{u \in X} \{v : (v,u) \in E \wedge \lambda((v,u)) = l \wedge \mu((v,u),k) = s\} \tag{14.7}$$

where $X \subseteq V$ is a set of nodes. OUT and IN return all reachable nodes with the label l and property with key k and value s. The OUT function

follows the direction of the edge and the IN function is a backward iteration of the edge.

We use following functions:

$$Filter_p(X) = \{v \in X : p(v)\} \tag{14.8}$$

$$Match_p(X) = Filter_p \circ TNodes(X) \tag{14.9}$$

$$Type_s(X) = TypeNode \circ Filter_{p_s} \circ TNodes \tag{14.10}$$

$$Stmt(X) = Statement(X) \tag{14.11}$$

The $Filter$ function returns all nodes of the set X that match the Boolean predicate $p(v)$. $TNodes$ is defined as a reusable traversal from the root of the Abstract Syntax Tree to all nodes. The $Match_p$ uses the $TNodes$ function to traverse all Abstract Syntax Tree nodes and only returns the nodes that match the filter function. The $Type_s$ function iterates the children of the Abstract Syntax Tree starting from node $x \in X$ searching for nodes of the type s. The $Statement(X)$ functions iterates the parents nodes until it reaches a statement node. This is important to get the statement where a specific node $x \in X$ is used. We use the short name $Stmt$ instead of $Statement$.

14.3 Methodology

The goal of Insecurity Refactoring is to inject vulnerabilities with different source code patterns into existing projects. This approach is based on static code analysis concepts. Figure 14.1 shows the process to inject vulnerabilities. The Code Property Graph [211] is used as an initial analysis model. Rules defined in the next section are applied to traverse the Code Property Graph to create the Adversary Controlled Input Dataflow (ACID) tree. The ACID tree is a tree representation of a backward data flow analysis. In the tree, a path from a leaf to the root represents data flow from a source to a sink. It is used as another analysis model to find Possible Injection Paths (PIP) or vulnerabilities. A vulnerability is basically a path (leaf to root) in the tree that does not contain any sanitization functions. In contrast, a PIP does contain a sanitization function. A PIP can be transformed to fit a vulnerability definition. For example, a sanitization method can be refactored into an insufficient sanitization method. To define insufficient sanitization methods, the context of the input data is analyzed using the context rules. The modifications are based on source code patterns. These patterns are defined in the PL/V pattern language that is described in section 14.6.1. Additional source code patterns can be injected by using the data flow

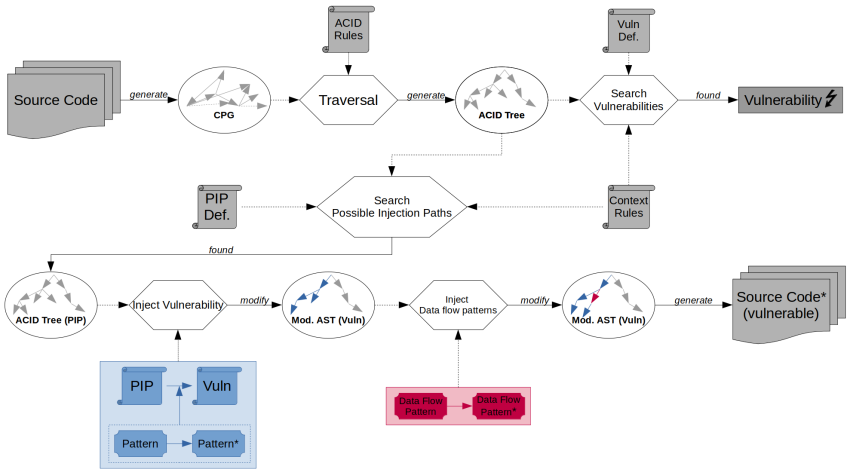


Figure 14.1: Overview of the Insecurity Refactoring process by using an ACID tree.

patterns to add some diversification. The ACID tree uses a tree structure based on nodes from the Abstract Syntax Tree. The Abstract Syntax Tree is an abstract representation of the source code. Refactoring is applied to the Abstract Syntax Tree to create a modified Abstract Syntax Tree. In the last step, the modified Abstract Syntax Tree is used to generate the insecurity-refactored source code.

14.3.1 Adversary Controlled Input Dataflow Tree

The first step for Insecurity Refactoring is to find PIPs. A PIP is a set of source code statements that can be refactored to inject a vulnerability. This approach focuses on vulnerabilities that have tainted data controlled by an adversary flowing from a source to a sink. Examples of vulnerabilities for this type are Cross Site Scripting (CWE-79), SQL Injection (CWE-89), Buffer Overflow (CWE-119), etc. Our approach uses a modified backward taint analysis. A normal taint analysis stops and removes any data that reaches sanitization methods. The modified taint analysis does not remove such data, instead it further tracks the data. This allows, in the refactoring step, to remove or modify the sanitization method to inject a vulnerability.

Data flow analysis can either be done forward (from source to sink) or backward (from sink to source). We use a backward data flow analysis using a Code Property Graph and follow specified rules to create an Adversary Controlled Input Dataflow tree. The idea is to use a backward data flow

analysis following each path of data that flows into the initial sink. These paths are represented in a tree, where the root is the sink of a vulnerability. Each leaf represents data that can reach the sink. Accordingly, a leaf represents a source. An advantage of creating a tree using backward data flow analysis is that the ACID tree allows analyzing all data concatenations that reach a sink. Every leaf represents possible input to reach the sink, but it doesn't necessarily mean that all of the leaves are concatenations.

Definition 3. An *Adversary Controlled Input Dataflow* (ACID) tree

$$T_{AC} = (V_{AC}, E_{AC}, \lambda_{AC}, \mu_{AC}) \tag{14.12}$$

is an ordered, rooted, directed, edge-labeled and attributed out-tree [188]. The nodes of the tree are defined in the set $V_{AC} \subseteq V_A$. Accordingly, the tree is based on Abstract Syntax Tree nodes and each node can be used to access the corresponding abstract syntax sub tree G_A from the Code Property Graph. The directed edges are defined as the set $E_{AC} \subseteq (V_{AC} \times V_{AC})$. The edge label function $\lambda_{AC} : E \to \Sigma_{AC}$ uses the alphabet Σ_{AC} to represent all edge names. The properties for the nodes are defined by the function $\mu_{AC} : V_{AC} \times K_{AC} \to S_{AC}$. The set K_{AC} defines the keys and the set S_{AC} defines the values. All attributes from the initial Abstract Syntax Tree nodes are found in the ACID tree nodes. Because the tree is ordered, similar to the Code Property Graph, we add an attribute with the key *childN* that stores the child position as a value.

The **root node** of an ACID tree is defined by:

$$\begin{aligned} root_{AC} &= root(V_{AC}, E_{AC}) \\ &= u \in V_{AC} : \nexists(\bullet, u) \in E_{AC} \end{aligned} \tag{14.13}$$

It returns the node, to which there are no directed edges pointing.

Additionally, to get all **leaf nodes** of an ACID tree, we define the following function:

$$\begin{aligned} L_{AC} &= L(V_{AC}, E_{AC}) \\ &= \{v \in V_{AC} : \nexists(v, \bullet) \in E_{AC}\} \end{aligned} \tag{14.14}$$

It returns all nodes that do not have an edge pointing to other nodes.

The **children** of a node v can be retrieved with the following definition:

$$C(v) = \{u : (v, u) \in E_{AC}\} \tag{14.15}$$

It returns a completely ordered set. The order is defined by the children positions in the tree.

Also, the **parent** of a node u can be retrieved with the following function:

$$p(u) = v \in V_{AC} \text{ where } (v, u) \in E_{AC} \tag{14.16}$$

The function

$$Path(l) = \langle l \rangle^\frown Path(p(l)) \tag{14.17}$$

defines a sequence of nodes starting from the leaf node l going upwards until reaching the root node $root_{AC}$ of the ACID tree.

The siblings of a node can be retrieved with the following functions:

$$Sib(v) = \{u : p(v) = p(u) | u \neq v\} \tag{14.18}$$
$$Bef(v) = \{u : p(v) = p(u) | \mu(u, childN) < \mu(v, childN)\} \tag{14.19}$$
$$Aft(v) = \{u : p(v) = p(u) | \mu(u, childN) > \mu(v, childN)\} \tag{14.20}$$

This allows to get all siblings (Sib), the siblings before (Bef), or the siblings after (Aft) a node v.

The edge labels are used to specify the data type that flows from one node to another. The data types are defined in the alphabet $\Sigma_{AC} = \{String, Numeric, Array, Unknown\}$. The properties for the nodes are defined in τ. We use the properties to define the different splits in the ACID tree. A split in the ACID tree means that either data from all sub trees will reach the sink, or only one sub tree at a time can reach the sink. The *link* property defines the link between children. The values are from the alphabet $\tau = \{\wedge, \oplus\}$. *Excluding* is defined by the symbol $\oplus$. It means that either one of the sub trees will reach the sink. A *concatenation* is defined by the symbol $\wedge$. It indicates that a concatenation of the children will reach the sink. We use the *cap* symbol, since every child has to add its input to the concatenation. For *excluding*, we assume that both paths are reachable in certain instances. Control statements decide which sub trees will reach the sink, i.e., they check code reachability.

14.3.2 Code example

Figure 14.2 shows a code example that is used to describe the process of Insecurity Refactoring. On line 7, the *getParam()* function is used to request the page number from the user. The page number is checked for being numeric (line 14) and sanitized using the *intval* (line 15) function. Hence, no Cross Site Scripting attacks are possible.

```
1  <?php
2  function getParam($param){
3    return $_GET[$param];
4  }
5
6  function page($debug, $name){
7     $page=getParam('page');
8
9
10
11
12
13
14    if(is_numeric($page)){
15       $out = $name . intval($page);
16       $out = "<a href='www.url.com/" . $out . "'> link </a>";
17    }
18    else {
19       $out = "Unknown page";
20    }
21
22    echo $out;
23 }
24 ?>
```

Figure 14.2: Code example shows proper sanitization (line 14 and 15) to prevent Cross Site Scripting.

14.4 ACID Tree Construction

The ACID tree is constructed by traversing the Code Property Graph. An ACID tree is created for each potential sink. The traversal requires a stack $stack_{call}$ that is used to correctly resolve function calls. The stack $stack_{call}$ stores the function calls that are resolved by the traversal. The traversal is based on different node types. The main traversal in the Code Property Graph is over the V_P nodes from the program dependence graph. We define the following node categories that are used by the Abstract Syntax Tree G_A and the program dependence graph G_P:

$$V_{assign} = \{v \in V_P \mid v \in V_A \mid \mu_A((v,u), type) = assignment\} \tag{14.21}$$

$$V_{param} = \{v \in V_P \mid v \in V_A \mid \mu_A((v,u), type) = parameter\} \tag{14.22}$$

Edges (E_P) point from a variable definition to statements (V_P) where the defined variable is used. Because the traversal is backwards, the definitions are traversed. A definition can either be an assignment ($v \in V_{assign}$) or it can be a function parameter ($v \in V_{param}$).

For the Control Flow Graph, the following node categories are important:

$$V_{function} = \{v \in V_C \mid v \in V_A \mid \mu_A((v,u), type) = function\} \tag{14.23}$$

The traversal also traverses concatenations, variables, function calls and coding constructs. We define the following sets to represent these node categories:

$$V_{exp} = \{v \in V_A \text{ that represent all expression}\} \tag{14.24}$$

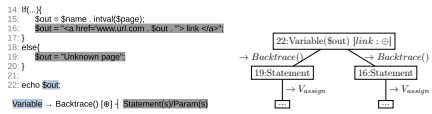
$$V_{var} = \{v \in V_{expr} | \mu_A(v, type) = variable\} \tag{14.25}$$

$$V_{con} = \{v \in V_{expr} | \mu_A(v, type) = concatenation\} \tag{14.26}$$

$$V_{call} = \{v \in V_{expr} | \mu_A(v, type) = call\} \tag{14.27}$$

$$V_{code} = \{v \in V_{expr} | v \text{ is a coding construct}\} \tag{14.28}$$

The set V_{exp} contains all expressions. An expression always has a return value. The traversal distinguishes expressions between variables V_{var}, concatenations V_{con}, function calls V_{call} and coding constructs V_{code}. For simplicity, all different concatenations that exist are unified by having the type *concatenation*. Coding constructs are different constructs that depend on the programming language. For example, in our implementation, the array attribute access is included. In the Abstract Syntax Tree, it is represented by a *dimension* node. The traversal is based on the different categories. For each powerset $\mathbb{P}$ of a



(a) Source code. (b) ACID tree.

Figure 14.3: Rules example of *Backtrace*() using the program dependence graph.

category, the corresponding function is used to traverse the Code Property Graph. The following function

$$Pos : V_A \to \mathbb{N} \tag{14.29}$$
$$Pos(v) = p \in \mathbb{N} | \mu(v, childnum) = p$$

allows getting a position of parameter. The position n can be used to get an expression of a function call with the following function:

$$CallExp(V, n) = OUT^{childnum,n}_{parent\_of}(V) \tag{14.30}$$

These functions are used in the ACID tree construction to correctly resolve function calls.

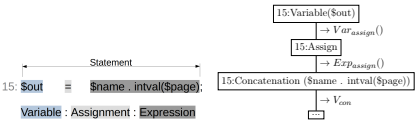
The following function defines **backward traversal** for the Code Property Graph:

$$Backtrace : \mathbb{P}(V_{var}) \to \mathbb{P}(V_{assign} \cup V_{param}) \tag{14.31}$$
$$Backtrace(V) = \bigcup_{v \in V} \{IN^{variable,v}_{reaches}(Stmt(\{v\}))\}$$

It uses the variable as input and returns the corresponding nodes where the variables are defined. It uses the *Stmt* function to get the statement where the variable v is used. The statement is used to get possible definitions of the variable v. The results can be assignments or parameters. Figure 14.3 shows how the *Backtrace*() function is used from the variable *$out*. In the ACID tree, these statements are added as children and the $\oplus$ defines that the children are mutually excluding. Accordingly, only one of the sub trees can reach the sink. For the variable, possible definitions are on line 19 and line 16.

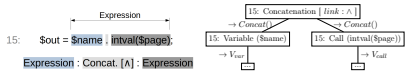
Based on the resulting statement type, the graph is traversed differently. If the statement is an assignment (V_{assign}), the following traversal rules apply to **resolve assignments**:

$$Assign : \mathbb{P}(V_{assign}) \to \mathbb{P}(V_{exp}) \tag{14.32}$$
$$Assign(V) = OUT^{childnum,1}_{parent\_of}(V)$$



(a) Source code. (b) ACID tree.

Figure 14.4: Rules example of an assignment.



(a) Source code. (b) ACID tree.

Figure 14.5: Rules example of a concatenation.

The Abstract Syntax Tree G_A is an ordered tree and the attribute *childnum* is used to define the order. The child number one is the expression that will be assigned. Because the ACID tree is constructed by a backward data flow analysis, the defined variable is added to the ACID tree first, then the assignment statement is added that is followed by the expression from $Assign()$. The defined variable is given by the OUT function using the child number zero.

Figure 14.4 shows how the rules are applied to line 15. Variable *$out* is added first, it is followed by the assignment node ($v \in V_{assign}$) which is followed by the expression.

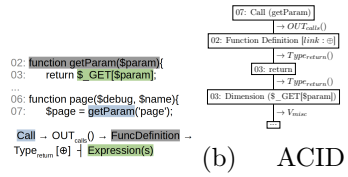
The following rules apply to **resolve concatenations**:

$$\begin{aligned} &Concat : \mathbb{P}(V_{con}) \rightarrow \mathbb{P}(V_{exp}) \\ &Concat(V) = OUT_{parent\_of}(V) \end{aligned} \tag{14.33}$$

The function $Concat$ allows to get the concatenated elements. Concatenations in the Abstract Syntax Tree use the children as operands. Accordingly, these are the elements that are concatenated and are provided by the OUT function.

Figure 14.5 shows how the $Concat$ function is applied to the top expression that is assigned to the variable $\$out$. The dot symbol is the standard method for string concatenation in PHP. A concatenation means that all inputs reach the sink in a concatenated form. Accordingly, the $\wedge$ symbol is added to the concatenation.

A function call has to be resolved to see if the function passes data from the input (parameter) to the output (return). The Code Property Graph already resolves function calls in G_C by the edges with the label *calls*. The



(a) Source code. (b) ACID tree.

Figure 14.6: Rules example of a function call.

following rules apply to **resolve function calls**:

$$CallReturn : \mathbb{P}(V_{call}) \rightarrow \mathbb{P}(V_{exp}) \tag{14.34}$$
$$CallReturn(V) = \bigcup_{v \in V} \{Type_{return}(OUT_{calls}(\{v\}))\}$$
$$\hookrightarrow \text{Side-effect: add call to stack } stack_{call}$$

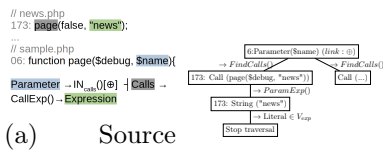
The OUT_{calls} returns the function definitions found in the control property graph. Because the traversal is a backward data flow analysis, further analysis has to continue on the output of the function (return statements). The $Type_{return}$ function finds all return statements inside a function definition. A combination of both functions $Call$ and $Type_{return}$ is used in $CallReturn$ to resolve function calls for the ACID tree. The resolving function call is put on the stack $stack_{call}$ to correctly resolve parameter expressions.

Figure 14.6 shows how the $CallReturn$ function is applied to the source code example. The function call *getParam* is passed to the $CallReturn$ function. It uses the edges E_C from the Control Flow Graph to find the corresponding function definition of the call on line 2. The $Type_{return}$ function is used to find all possible return statements (line 3) of the function definition. If more than one return statement is found, multiple returns are added as mutually excluding ($\oplus$).

For each of these coding constructs, different approaches are required to correctly continue the traversal. The following definitions **resolve coding constructs**:

$$Code : \mathbb{P}(V_{code}) \rightarrow \mathbb{P}(V_{exp}) \tag{14.35}$$
$$Code(V) =$$
$$\bigcup_{v \in V} \bigcup_{p \in P_{code}} \{ASTNode_{in}(p, v), \text{ if } Match_p(v)\}$$

Because different code constructs require different approaches to get the correct input, we use the PL/V pattern language described in section 14.6.1.



(a) Source code. (b) ACID tree.

Figure 14.7: Rules example of reaching a parameter with an empty call stack.

The set P_{code} contains all implemented code construct patterns. The $Match$ function is used to check if the Abstract Syntax Tree node v equals the pattern p. The $ASTNode$ function returns the corresponding input nodes of the Abstract Syntax Tree.

As previously stated, the $Backtrace$ function also returns parameters. Depending on whether a function call is currently resolved or not ($stack_{call} = \varnothing$), the backward data flow analysis has to traverse differently and is defined as follows:

$$Param : \mathbb{P}(V_{param}) \rightarrow \mathbb{P}(V_{exp}) \tag{14.36}$$
$$Param(V) = \bigcup_{v \in V} \begin{cases} FindCalls(\{v\}), \text{ if } stack_{call} = \varnothing \\ BackToCall(Pos(v)), \text{ otherwise} \end{cases}$$

It is a simple function that decides if the $FindCalls$ or $BackToCall$ is used to traverse parameters V_{param}. If the stack $stack_{call}$ is not empty, the function $BackToCall : \mathbb{N} \rightarrow V_{exp}$ jumps back to the initial function call found on top of the stack $stack_{call}$. The traversal is continued from the corresponding expression based on the parameter position.

If $stack_{call}$ is empty, the following rules are applied to **find all calls**:

$$FindCalls : \mathbb{P}(V_{param}) \rightarrow \mathbb{P}(V_{exp}) \tag{14.37}$$
$$FindCalls(V) = \bigcup_{v \in V} \{CallExp(IN_{calls}(\{v\}), Pos(v))\}$$

The $FindCalls$ function returns all function calls of the function from the parameter. It uses the function $Pos(p)$ to return the position of the parameter in the function definition. The CallExp function is required to continue the backward data flow analysis from the correct parameter of the function calls.

Figure 14.7 shows how a parameter is resolved when the stack is empty. The sample.php is from the code example and one call of function *page* is found in the news.php file on line 173. Additional function calls are added as mutually excluding ($\oplus$) children. The $ParamExp$ function uses the parameter position to get back the correct expression of the function call.

In the example, the parameter is in the second position. Accordingly, the second expression in the function call is returned that is the string literal *"news"*. Because a literal is not found in any of the categories except of V_{exp}, the traversal stops here.

Some function calls can be resolved by the control property graph. Other functions that pass data from a parameter to the return value are not resolved by the control property graph. For example, the *intval* function will not be resolved by the control property graph. To solve that problem, we define the pair $V_{pass} = (F, \delta)$. The set F contains all functions that return data from parameters. The mapping from input to output is defined by the set $\delta \subseteq (F \times \mathbb{N}_0)$. The output of a passthrough function is always the return value. Inputs are parameters that are referenced by the parameter position by a natural number $\mathbb{N}_0$. If the *CallReturn* is unable to resolve the call, the following function will **resolve passthrough** functions:

$$Passthrough : \mathbb{P}(V_{call}) \rightarrow \mathbb{P}(V_{exp}) \tag{14.38}$$
$$Passthrough(V) = \bigcup_{v \in V} \{CallExp(\{v\}, n) | (v, n) \in \delta\}$$

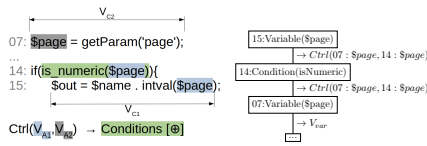
The function returns the corresponding expression based on the parameter position n.

14.4.1 Control functions

If we look into the code sample in figure 14.2, the security relevant function *is_ numeric* will not be traversed. This is a sanitization function call that changes the control flow without changing any data in the data flow. Accordingly, another step is required to find security relevant function calls that change the control flow. Such functions can occur in the traversal of the $Backtrace : \mathbb{P}(V_{var}) \rightarrow \mathbb{P}(V_{assign} \cup V_{param})$ function. For each result, the following functions allow to **find control statements**

$$Ctrl(var_{def}, var_{use}) = \tag{14.39}$$
$$\bigcup_{path \in Paths(var_{def}, var_{use})} \{ \bigcup_{v \in path} \{Filter_{if}(\{v\}) \circ Match_{var_{def}}(\{v\})\}\}$$

where the function *Paths* returns all possible paths in the Control Flow Graph G_C from variable definition v_{def} to variable usage v_{use}. The $Filter_{if}(c)$ filters only statements that are actually *if* statements. Additionally, the $Match_{var_{use}}(c)$ checks if the initial variable is used inside the *if* statement. Overall, the *Ctrl* function returns all *if* statements where the variable var_{def} is used. In the ACID tree, the control functions are added in between the variable usage and variable definition. The *if* statements themselves are also



(a) Source code. (b) ACID tree.

Figure 14.8: Rules example of finding condition checks.

parsed to correctly handle all conditions that are used. A union in the if statement is a mutually excluding split ($\oplus$) in the ACID tree because only one of the conditions has to be true. For a complement, both conditions are added in serial to the ACID tree because both of the conditions have to be true. Figure 14.8 shows how the *is_ numeric* is found by the condition check. The *Backtrace* function returns from the variable $*page* on line 15 the assignment on line 7. The *is_ numeric* uses the variable $*page* and is found in an *if* statement. Accordingly, it will be added to the ACID tree.

14.4.2 Data flow type

The construction of the ACID tree requires labeling the edges (λ_{AC}) corresponding to the flowing data type. The flowing data type can only be determined by a forward analysis. Accordingly, the labels are set after the initial ACID tree is constructed. The labels are defined by iterating the nodes of the paths ($Path(l)$) from all leaves of the ACID tree. If a node defines a data type change, the data type changes. For example, the *intval()* function changes the data type to *numeric*. In contrast, a string concatenation changes the data type to string. All nodes that do not change a data type will preserve the previous data type.

14.4.3 ACID tree example

Figure 14.9 shows the full ACID tree for the initial code example shown in figure 14.2. All leaves are data that can reach the sink. For each node the corresponding line number is added. An edge label represents the data type that flows between the nodes. A simple guide to read an ACID tree is to choose a leaf and go upwards until you reach the root. If you reach a concatenation, the symbol $\wedge$ is shown. It means that the data from the other sub trees of that $\wedge$ node are also included by a concatenation to the sink. In contrast, the $\oplus$ means that only either one of the sub trees reaches the sink. In that case, the other sub trees can be ignored.

In the example, either the *"Unknown page"* string or the *_GET* variable are concatenated ($\wedge$) with the $\$name$ parameter and the two string values will reach the echo function. The parameter can be different based on what function call is used. In the example, the parameter has the string value *"news"*. The condition functions *isNumeric* and *intval* prevent any Cross Site Scripting attacks.

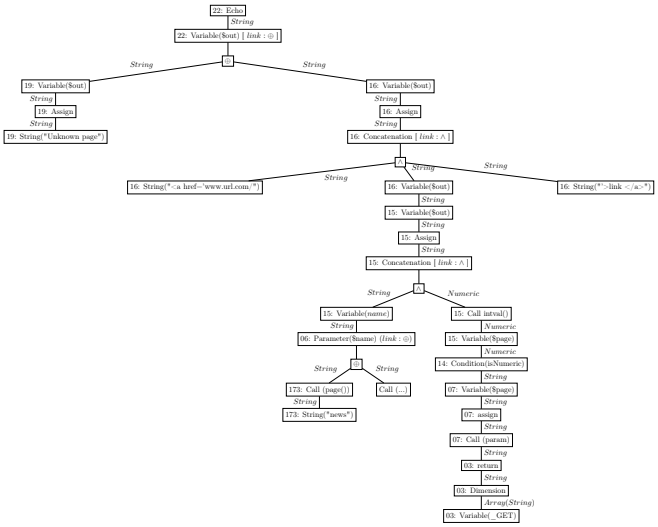


Figure 14.9: ACID tree of the code example from figure 14.2.

14.5 Insecurity Refactoring

Refactoring is defined as a change made to the internal structure of software to make it easier to understand and less expensive to modify without changing its observable behavior [191] [198] [199]. Insecurity refactoring uses a similar approach and we define it as: *Insecurity refactoring is a change to the internal structure of software to inject a vulnerability without changing the observable behavior in a normal use case scenario. If the injected vulnerability is exploited, the observable behavior will change.* Accordingly, insecurity refactoring requires to maintain the normal use of the program. The following rules define insecurity refactoring by transforming a PIP into a vulnerability.

14.5.1 Vulnerability Description

Vulnerabilities can be described in different ways. For example, Martin et al. [196] used the Program Query Language to describe vulnerabilities. Our

focus is on vulnerabilities that rely on data flow from a source to a sink. We define a vulnerability based on three sets V_{src}, V_{dst}, and V_{san}. The sources V_{src} are patterns for retrieval of tainted data. In the code example the \$_ GET global array in line 3 is included in the source pattern set (V_{src}) because it provides user-controlled data. The *echo* function in line 22 is a sink contained in the sinks set V_{dst}.

The ACID tree is based on a backward data flow analysis with a sink and an amount of inputs (sources). The inputs are represented by the leaves of the ACID tree.

Definition 4. A data flow path $dfp^l_{AC} = (T_{AC}, l)$ is defined as a pair containing the ACID tree and a chosen leaf.

The data flow path represents data that flows from the chosen leaf l into the sink r_{AC}.

The sanitization functions are defined by the set V_{san}. Sanitization functions from V_{san} depend on the context and the vulnerability type. The following function *Suff* allows to check if a sanitization function v_{san} is sufficient with respect to the data flow path dfp^l_{AC}:

$$Suff(v_{san}, dfp^l_{AC}) = \begin{cases} True, & \exists c : ((v_{san}, root_{AC}), c) \in S \\ & \text{and } c \in Context(dfp^l_{AC}) \\ False, & \text{otherwise} \end{cases} \tag{14.40}$$

where

$$S \subseteq ((V_{san} \times V_{dst}) \times C_{ctx})$$

The set C_{ctx} contains all possible context types. The set S defines for each sanitization function v_{san} and sink v_{dst} what context c is required. The function $Context()$ returns a set of all active contexts for the data flow path. The details of the function are described in section 14.6.2. It returns all contexts that are found in the data flow path dfp^l_{AC}. Accordingly, the function *Suff*() checks if the sanitization function is sufficient based on the vulnerability type.

Definition 5. The set $Vuln_{AC}$ contains vulnerabilities. A vulnerability $vuln^l_{AC}$ is a data flow path dfp^l_{AC} with the following properties:

$$root_{AC} \in V_{dst} \tag{14.41}$$

$$l \in V_{src} \tag{14.42}$$

$$\nexists v \in Path(l) : v \in V_{san} \wedge Suff(v, dfp^l_{AC}) \tag{14.43}$$

A vulnerability exists if tainted data from a leaf l (source) reaches the root $root_{AC}$ (sink) without passing any sanitization function from set V_{san}. For each vulnerability type, the source, sanitization functions and sinks are defined. The different sanitization functions defined in V_{san} are sufficient to prevent a vulnerability depending on the vulnerability type. For example, a sanitization function to prevent SQL Injection is usually not sufficient to prevent XSS attacks. The code example is not vulnerable because two sufficient sanitization methods for XSS are used.

14.5.2 Possible Injection Path

A possible injection path (PIP) is a data flow path in the source code that can be transformed into a vulnerability. The sets of sources and sinks are extended by additional sources and sinks that are usually secure to use. We define the set of PIP sources as $P_{src} \supseteq V_{src}$ and the set of PIP sinks as $P_{dst} \supseteq V_{dst}$. As an example, a secure source would be a function that only returns an integer value from the user. A secure sink could, e.g., be a bind query function from a parameterized SQL query.

Definition 6. The set Pip_{AC} contains possible injection paths. A possible injection path pip^l_{AC} is a data flow path dfp^l_{AC} with the following properties:

$$root_{AC} \in P_{dst} \tag{14.44}$$

$$l \in P_{src} \tag{14.45}$$

$$\exists v \in Path(l) : v \in V_{san} \wedge Suff(v, dfp^l_{AC}) \tag{14.46}$$

$$\vee l \notin V_{src} \vee root_{AC} \notin V_{dst} \tag{14.47}$$

The PIP definition is similar to a vulnerability definition. A PIP exists if data from a leaf node $l \in P_{src}$ (source) reaches the root node $root_{AC} \in P_{dst}$ (sink). At least one sanitization function ($p \in V_{san}$) in the path ($Path(l)$) has to be found or at least one secure source ($l \notin V_{src}$) or secure sink ($root_{AC} \notin V_{dst}$) has to be found. These requirements ensure that a PIP is not already a vulnerability. The nodes of the path from l to $root_{AC}$ are contained in $Path(l)$ and represent the data flow. That path will be used to transform the vulnerability with different source code patterns.

The code example is by definition a PIP. The path from $\_GET$ variable leaf $l \in P_{src}$ reaches the root node that represents the sink echo ($root_{AC} \in P_{dst}$). As requirements, the sanitization functions $intval()$ and $isNumeric()$ are found. The next section explains the required code changes to perform insecurity refactoring.

14.5.3 Injecting a vulnerability

The transformation of a PIP into a vulnerability uses the following transformation sets:

$$T_{src} \subseteq (P_{src} \times V_{src}) \tag{14.48}$$
$$T_{dst} \subseteq (P_{dst} \times V_{dst}) \tag{14.49}$$
$$T_{san} \subseteq (V_{san} \times V_{san}) \tag{14.50}$$

For the source transformation set T_{src}, secure source functions $p_{src} \in P_{src}$ are mapped to insecure functions $v_{src} \in V_{src}$. In the same manner, secure sinks $p_{dst} \in P_{dst}$ are mapped to insecure sinks $v_{src} \in V_{src}$. The sanitization functions are mapped to each other and depending on the *Suff* function that can be used to make the sanitization functions insufficient resulting in vulnerabilities. The sets T_{src}, T_{dst} and T_{san} only map functions that can be replaced with each other without breaking the insecurity refactoring definition.

The possible injection path pip^l_{AC} can be transformed into a vulnerability $vuln^l_{AC}$, if the following **condition** check holds:

$$Check : pip^l_{AC} \rightarrow Boolean \tag{14.51}$$
$$Check(pip^l_{AC}) = \bigwedge_{v \in Path(l)} \begin{cases} Ch_{src}(v), & \text{if } v = l \\ Ch_{dst}(v), & \text{if } v = root_{AC} \\ Ch_{san}(v), & \text{if } v \in V_{san} \\ True, & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} Ch_{src}(l) =& l \in V_{src} \vee \exists (l, l') \in T_{src} \\ Ch_{dst}(r) =& r \in V_{dst} \vee \exists (r, r') \in T_{dst} \\ Ch_{san}(v) =& \neg Suff(v, pip^l_{AC}) \\ & \vee \exists (v, v') \in T_{san} \wedge \neg Suff(v', pip^l_{AC}) \end{aligned}$$

The PIP condition check checks that for all ($\bigwedge$) nodes if it is a sufficient sanitization, secure sink or secure source that there exists an insecure representation. That ensures that the PIP can be transformed into a vulnerability. If this condition check returns *False* for a PIP pip^l_{AC}, it means that additional transformations in the transformations set are required to inject a vulnerability.

The transformation of a PIP into a vulnerability requires to modify the

ACID tree. The replacement function is defined as follows:

$$\begin{aligned} Replace_v^{v'}(T_{AC}) =& T'_{AC} \\ =& (V'_{AC}, E'_{AC}, \lambda'_{AC}, \mu'_{AC}) \end{aligned} \tag{14.52}$$

where

$$\begin{aligned} V'_{AC} =& V_{AC} \setminus \{v\} \cup \{v'\} \\ E'_{AC} =& E_{AC} \setminus \{(v,u)\} \cup \{(v',u)\} \\ & \setminus \{(w,v)\} \cup \{(w,v')\} \\ (v,u) \in& E_{AC} \\ (w,v) \in& E_{AC} \end{aligned}$$

The *Replace* function replaces a node v in an ACID tree T_{AC} with the node v'. It requires to connect the old edges that point to and from v to the replaced node v'.

Definition 7. A possible injection path pip^l_{AC} that passes the condition check can be transformed into a vulnerability $vuln^l_{AC}$ with the following function:

$$\begin{aligned} Tf : Pip_{AC} \rightarrow& Vuln_{AC} \\ Tf(pip^l_{AC}) =& (G'_{AC}, l') \\ l' =& Tf_{src}(l) \\ G'_{AC} =& (Replace_l^{Tf_{src}(l)} \circ Replace_{root_{AC}}^{Tf_{dst}(root_{AC})} \\ & \bigcirc_{v \in Path(l)} Replace_v^{Tf_{san}(v)})(G_{AC}) \end{aligned} \tag{14.53}$$

where

$$Tf_{dst}(r) = \begin{cases} r, & \text{if } r \in V_{dst} \\ r' : (r, r') \in T_{dst}, & \text{otherwise} \end{cases}$$

$$Tf_{src}(l) = \begin{cases} l, & \text{if } l \in V_{src} \\ l' : (l, l') \in T_{src}, & \text{otherwise} \end{cases}$$

$$Tf_{san}(v) =$$

$$\begin{cases} v, & \text{if } v \notin V_{san} \\ v, & \text{if } \neg Suff(v, pip^l_{AC}) \\ v' : (v, v') \in T_{san} \wedge & \neg Suff(v', pip^l_{AC}), \text{ otherwise} \end{cases}$$

The ring operator ($\circ$) represent that all those functions have to be processed to transform a PIP into a vulnerability. In words, the transformation

is done by replacing a secure sink ($l \notin V_{dst}$) by an insecure sink ($l \in V_{dst}$). In the same manner, a secure source is replaced by an insecure source. Additionally, all the sanitization functions in the path from source to sink have to be replaced by insufficient sanitization methods. In the code example, the functions $isNumeric$ and $intval$ ($\in V_{san}$) have to be replaced by insufficient sanitization methods ($\notin V_{san}$).

14.6 Implementation

This section describes the implemented PL/V pattern language that is used to detect and inject source code patterns to transform a PIP into a vulnerability.

14.6.1 The PL/V pattern language

Source code patterns are described in the PL/V language. PL/V is a context-free language that can be described in BNF as shown in figure 14.10.

A *Pattern* consists of multiple code lines. If it is only a single code line, it can be an expression or a statement. Multiple code lines represent a statement list in which each line must be a statement. A code line has an identifier *id* and a parameter list *PrmList*. The identifier represents a language pattern. Language patterns are used to decouple the source code patterns from specific programming languages. For example, the pattern <=> ($\%in, \%out$) represents an assignment. It uses the symbol = as an identifier. Accordingly, a language pattern exists for the id =. The language patterns contain the information on how the Abstract Syntax Tree representation of a specific language pattern looks like. This allows to generate the Abstract Syntax Tree of the language patterns.

The parameter list can contain other patterns, literals, variables and any nodes. The variables input ($\%in$) and output ($\%out$) are special case variables that can be used to chain source code patterns. Additionally, this allows to get input or output nodes of source code patterns by using the $ASTNode_x$ function.

In the example, the input is the expression that represents the value of the assignment. The output is the variable that will be assigned to.

Source code patterns can contain $<$ *any* $>$ parameters with optional ("?") and multiple suffixes. An "any" parameter means that the parameter can be anything. The optional suffix ("?") specifies that the parameter does not have to exist. The "multiple" suffix ("...") specifies that any number of parameters can occur, but at least one parameter has to exist. A combination

```
<Pattern>   ::= <CodeLines>
<CodeLines> ::= <CodeLine> "\n" <CodeLines>
              | <CodeLine>
<CodeLine>  ::= "<" <id> ">" <PrmList> ")"
<PrmList>   ::= <Param> "," <PrmList>
              | <Param>
<Param>     ::= <Pattern>
              | "%var" | "%in" | "%out"
              | literal
              | <Any>
<Any>       ::= "<any>" | "<any>..." | "<any>?"
              | "<any>?..."
```

Figure 14.10: BNF of PL/V language.

means that any number of parameters can occur including none. Literals are fixed values that are used in the corresponding pattern.

All patterns must contain a input $\%in$ and output $\%out$ variable except of a source and sink pattern. A pattern representing a source has only the fixed output variable. In contrast, a pattern of a sink only has the fixed input variable. Other variables can be used inside the pattern. For example, if a pattern requires accessing a specific key of an array, the key can be set as a variable $\%var$ and be used in further patterns.

The following example stems from our patterns [193]. It defines the sanitization pattern representing *htmlspecialchars*:

$$< call > (htmlspecialchars, \%in, < any > ())$$

The pattern uses the *call* language pattern. The *call* pattern requires a literal (*htmlspecialchars*) to define the function name in the Abstract Syntax Tree. The $\%in$ defines the input parameter. The *htmlspecialchars* function has an optional parameter that is represented as $< any >$. The output is the return value of the function.

For ACID tree construction, different source code patterns are required. For example, concatenations and the coding constructs are represented in the PL/V language. To find source code patterns it is required to have an equal check for each pattern. The following function allows to check if a part of the Abstract Syntax Tree from the Code Property Graph matches the pattern:

$$Match_p(v) = \tag{14.54}$$
$$\begin{cases} False, & \text{if } \neg Type_p(v) \\ True, & \text{if } p = \emptyset \\ \bigwedge\limits_{p_i, v_i \in C_{pat}(p,v)} Match_{p_i}(v_i), & \text{otherwise} \end{cases}$$

The $Type_p(v)$ function checks if the type of the Abstract Syntax Tree node v equals the type of pattern node p. The $C_{pat}(p, v)$ function returns pairs of the children from the AST node v and pattern node p. $Match_p(v)$ checks recursively if the pattern node type is the same as the node type from the Abstract Syntax Tree. Parameter v represents the root node of the Abstract Syntax Tree that is checked and p is the root node of the pattern Abstract Syntax Tree. For simplicity, the *any* nodes are not specified in the $Match_p$ function. It simply checks based on the suffixes if the corresponding parameters exist or not.

The different variable nodes $(\%in, \%out, \%var)$ are used to represent important nodes. They are defined in the PL/V language. The following function allows to get the corresponding node in the Abstract Syntax Tree G_A based on the variable node x:

$$ASTNode_x(p, v) = \tag{14.55}$$
$$\begin{cases} v, & \text{if } p = x \\ \bigcup\limits_{p_i, v_i \in C_{pat}(p,v)} ASTNode_x(p_i, v_i), & \text{otherwise} \end{cases}$$

It is a recursive function that searches for the same position in the Abstract Syntax Tree of the Code Property Graph starting from node v as in the sub tree of the pattern starting from node p.

14.6.2 Context analysis

The ACID tree is an analysis model that is used to evaluate the context. For each data flow path dfp^l_{AC} the context can be specified. A context c can be identified by what is concatenated before the input (*pre*) and what is concatenated after the input (*post*). Instead of formalizing the context check, we define for each context ($c \in C$) the function $IsContext_c(pre, post)$. The inputs *pre* and *post* are both string values. The function returns a boolean value and uses different checks to specify if the context c exists for the inputs.

We define the following function to get a set of all contexts for a data

flow path dfp^{l}_{AC}:

$$Context(dfp^{l}_{AC}) = \{c \in C_{ctx} \text{ and } IsContext_c(Up_{pre}(l), Up_{post}(l))\} \tag{14.56}$$

It uses the recursive functions $Up_{pre}(l)$ and Up_{post} to get the string values that the input is concatenated with. These functions are defined as follows:

$$Up_{pre}(v) = \begin{cases} \bigcup\limits_{c \in Bef(v)} Down(p(c)), & \text{if } p(v) = \wedge \\ Up_{pre}(p(c)), & \text{otherwise} \end{cases} \tag{14.57}$$

$$Up_{post}(v) = \begin{cases} \bigcup\limits_{c \in Aft(v)} Down(p(c)), & \text{if } p(v) = \wedge \\ Up_{post}(p(c)), & \text{otherwise} \end{cases} \tag{14.58}$$

Up is a recursive function that iterates from the leaf upwards to the root node. If the parent of node v is a concatenation ($\wedge$), data will be concatenated to the input data. Accordingly, that concatenated data is the context of the input data. Based on *pre* or *post* context, the corresponding siblings before or after of the input nodes are analyzed using the *Down* function. In the initial code example, on line 15 there is a concatenation of the variable $\$name$ and the variable $\$page$. The variable $\$page$ is the input from the user and the variable $\$name$ is the context that is concatenated.

The function

$$Down(v) = \begin{cases} String(v) & \text{if } Type_{string}(v) \\ \bigcup\limits_{c \in C(v)} Down(c), & \text{if } v = \wedge \\ Down(First(v)), & \text{if } v = \oplus \\ Down(C(v)), & \text{otherwise} \end{cases} \tag{14.59}$$

is a recursive function that iterates the tree downwards. If a concatenation is found, the recursive function of the children nodes will be united. A problem may occur if an excluding node $\oplus$ is reached. In the example on line 6, the children from $Parameter(\$name)$ are excluding. The downwards recursive function has to decide which mutually excluding child will be used for the context analysis. Different approaches can be used. Either one child is selected and used for the context analysis or all children are checked to see if they result in a similar context. We decided to use the context of the first child. It is a simple heuristic under the assumption that the context will not differ from other sub trees. Even if the context is different based on the different sub trees, at least one sub tree has the analyzed context. Accordingly, the approach can only ensure the chosen case will be exploitable.

The context analysis of the ACID tree sample in figure 14.9 shows that the user-provided data is concatenated with the String *news* and *link *. Accordingly, the output on the web page will be:
*link *.

In the code example, there is a potential Cross Site Scripting sink. Accordingly the Cross Site Scripting relevant context checks are required. In the example, the context check for HTML attribute context and *inside apostrophes* context will return true.

14.6.3 Insert data flow pattern

A main goal of insecurity refactoring is to create learning examples. Previous research [205] [206] showed that many interesting source code patterns are data flow source code patterns. The path $Path(l)$ defines the nodes from a source to a sink. It also represents the data flow of the PIP. Depending on what kind of learning example should be created, different data flow patterns are interesting. For example, some data flow patterns are difficult to detect by static code analysis tools [207]. If the learning examples should be more focused on *Capture the flag* (CTF) events, data flow patterns can be added that, for example, teach specific techniques like dynamic function calls. Also data flow patterns can be used to make the vulnerability difficult to detect by dynamic analysis tools (e.g. fuzzers).

Definition 8. The transformation of data flow patterns is defined as a tuple:

$$T_{df} = (D, M, \mu_R) \tag{14.60}$$

The set D defines all data flow patterns. The set $M \subseteq (D \times D)$ defines the patterns that can be replaced with other data flow patterns. The function $\mu_R : D \to R$ defines what requirements $r \in R$ are required for the inserted data flow pattern $d \in D$. The set R contains all requirements. A requirement $r \in R$ is a combination of a context $c \in C_{ctx}$ and a boolean that defines if the context must exist or must not exist. If all requirements for a data flow pattern are fulfilled, the data flow pattern can be injected without breaking the insecurity refactoring definition.

Usually the patterns that used to be replace with interesting data flow patterns are simple like an assignment. Interesting patterns represent different difficulties of the vulnerabilities. The requirements $r \in R$ have to be fulfilled to transform the source code maintaining the insecurity refactoring requirements. For example, one data flow pattern will redirect to the main

```
1 <def_func>(sanitize, <param_list_1>(<$>(a)),
      <stmtlist>(<return>(<$>(a))))
2 <=>(<$>(func), <s>(sanitize))
3 <=>(%out, <call_v>(func, %in))
```

Figure 14.11: Function call by string described in the PL/V language.

page if the tainted variable is not an integer. The pattern does not contain an exit statement and the source code later on is still executed (Pattern found in CVE-2013-3524). This pattern requires that the initial PIP contains a restriction to integer only variables. The requirement ensures that the program will still run as normal as long as only integer values are inserted. But it will change its external behavior as soon as attackers insert unintended values like a string. The patterns are searched in the path $Path(l)$ using the $Match_p(n)$ function.

14.6.4 Source code modification example

All the code modifications are based on the transformation sets (T_{src}, T_{dst}, T_{san} and T_{df}). Each element of the sets can define different variables that are required to perform a code modification. The modifications are done on the Abstract Syntax Tree (G_A). The variables of the PL/V language represent sub trees of G_A. Figure 14.12a shows the code example and figure 14.12b shows the insecurity-refactored source code. Applying the rules to inject a vulnerability requires to replace the *intval* function and $is\_numeric$ with an insufficient sanitization pattern $p \in T_{san}$. The source and sink do not require any modifications. As described previously, data flow patterns allow to introduce different source code patterns. In the example, the assignment ($<=>(\%out, \%in)$) pattern on line 7 is used to introduce a data flow pattern. For that assignment the output $\%out$ is the AST sub tree that represents the variable *page*. For the input $\%in$ the correspding AST sub tree represents the expression that is assigned to the variable. In that case, it is the function call *getParam*. Figure 14.11 shows the inserted data flow pattern. This pattern is difficult for static code analysis tools [207]. The insecurity-refactored source of the PIP is shown in figure 14.12b. On line 14, the sanitization function $is\_numeric$ is replaced by the insufficient sanitization function is_string. On line 15, the sanitization function *intval* is replaced by the *htmlspecialchars* sanitization function. The sanitization function is insufficient for the *inside apostrophes* context. It would require an *inside quotes* context to be sufficient. A potential Cross Site Scripting attack could inject a *onclick* parameter with Javascript payload. Lines 7 to

12 show the source code pattern that is difficult for static code analysis tools. It represents a dynamic function call using a string value.

All the modifications to inject a vulnerability are done by modifying the Abstract Syntax Tree. The next step is to revert the modified Abstract Syntax Tree back into actual source code. A simple program was written that generates PHP source code based on the Abstract Syntax Tree. An Abstract Syntax Tree uses some kind of abstraction to unify functions with the same functionality. For example, *<?=* and *echo* are both represented by an *echo* function call. The current approach checks the Abstract Syntax Tree to determine what lines of code are modified. Only the modified lines are replaced by the generated PHP source code and other lines of the files are maintained. This diminishes the chance that an abstraction breaks the source code.

The injection of vulnerabilities is semi-automated. For each PIP, the tool shows the critical sanitization functions that have to be replaced. The tool provides a list of sanitization functions that are insufficient for the corresponding context. In addition, patterns can be selected to be injected. After selecting the injected patterns, the tool checks if all sanitization functions will be replaced with insufficient sanitization functions. If all of them are selected, the vulnerability will be injected. For a fully automated approach, the patterns could be selected randomly.

```php
<?php
function getParam($param){
  return $_GET[$param];
}

function page($debug,
    $name){
   $page=getParam('page');









   if(is_numeric($page)){
      $out = $name .
          intval($page);
      $out = "<a
          href='www.url.com/"
          . $out . "'> link
          </a>";
   }
   else {
      $out = "Unknown
          page";
   }

   echo $out;
}
?>
```

(a) The original code example.

```php
<?php
function getParam($param){
  return $_GET[$param];
}

function page($debug, $name){
   function sanitize($a)
   {
     return $a;
   }
   $func = "sanitize";
   $page =
       $func(getParam("page"));

   if(is_string($page)){
      $out = ($name .
          htmlspecialchars($page));
      $out = "<a
          href='www.url.com/" .
          $out . "'> link </a>";
   }
   else {
      $out = "Unknown page";
   }

   echo $out;
}
?>
```

(b) Insecurity-refactored code example.

Figure 14.12: Insecurity refactoring using a data flow pattern that is difficult for static code analysis tools (function call by string).

14.7 Evaluation

The evaluation explores whether the Insecurity Refactoring approach is applicable to real projects. Additionally, it is important to see if the insecurity-refactored projects break the Insecurity Refactoring definitions. The main condition is that the projects can still be executed for normal use. Also the usage as learning example is evaluated.

14.7.1 Open source projects

We developed a tool to perform insecurity refactoring on PHP projects [193]. First of all, we want to see if insecurity refactoring can be used to inject vulnerabilities in open source projects. This requires to define the set of sources, sinks, etc. Table 14.1 shows how many entries are in each set. We retrieved the sets by reviewing the PHP documentation [201]. Each sanitization function from V_{san} that passes through data is also found in the passthrough data set (V_{pass}). The other passthrough functions are mainly functions to manipulate string values. The SQLi sinks contain different functions because each database has different PHP drivers. We added all functions from SQL database drivers we found in the PHP documentation. Only 9 sources are in our data set that are mainly functionality from PHP like the global array *_GET*. *Eval* and *unserialize* are different vulnerability types. Each of them is represented as a sink for their category.

A crawler tool was written that crawls GitHub [192] for projects that contain PHP source code. The corresponding source code is then checked for PIPs. Table 14.2 shows the results. 307 open source projects were scanned. In 25 of these projects PIPs were found. Accordingly, the tool could inject vulnerabilities in 8.1% of the projects. It also shows that most of the PIPs are related to Cross Site Scripting. Not many projects contained PIPs related to textiteval or textitunserialize. We also found several vulnerabilities. Those reports were reviewed: 55 true vulnerabilities and 68 false positive reports. Most of the vulnerabilities were found in installation and testing

Type	Sources (P_{src})	Sanitization (V_{san})	Passthrough (V_{pass})	Sinks (P_{dst})
All	9	98	164	•
XSS	•	•	•	9
SQLi	•	•	•	77
Eval	•	•	•	1
Unserialize	•	•	•	1

Table 14.1: Possible injection path data set.

	PIP	True Positive (Vuln)	False Positive (Vuln)
XSS	221	16	57
SQLi	98	37	10
Eval	1	0	1
Unserialize	3	2	0
	323	55	68

Table 14.2: Possible injection paths found in 25 open source projects out of 307 scanned projects.

files or were deliberate vulnerabilities. Three vulnerabilities were potentially dangerous vulnerabilities in commonly used projects. These vulnerabilities were reported by us to developers and they confirmed and patched the vulnerabilities (CVE-2020-27163, CVE-2021-3318, CVE-2021-26716). The vulnerability reports also included 68 false positive reports. In these cases pre-conditions prevent an exploitation. These pre-conditions were outside the ACID tree and could not be detected. The false positive vulnerability reports show that not all sanitization approaches can be detected. Nevertheless, PIPs that are used for insecurity refactoring have to contain a detected sanitization function to inject a vulnerability. This decreases the possibility that an injected vulnerability is not exploitable. One problem for Cross Site Scripting exists if the *Content-Type* is set to a secure type (e.g. *plain/text*). The ACID tree does not contain the *Content-Type*. If a PIP is found in a file that sets a secure *Content-Type*, the injected vulnerability will not be exploitable. If no sanitization function is found, but a special sanitization function exists, it will not be selected as as PIP. Accordingly, those false positive reports do not impact the possibility for insecurity refactoring.

The results demonstrate that the concept of insecurity refactoring works on open source projects. By increasing the data set of sources and sinks, the chances of finding PIPs can be increased. Adding support for object-oriented data flows to the initial Code Property Graph will increase the findings.

14.7.2 Learning examples

Initial evaluation shows that PIPs are found in open source projects. The next step is to see if the refactoring itself works without breaking the functionality of the projects. Additionally, it is important to check if the injected vulnerabilities can actually be used to teach software security. For the evaluation, two exercises were arranged for two different groups. The exercises

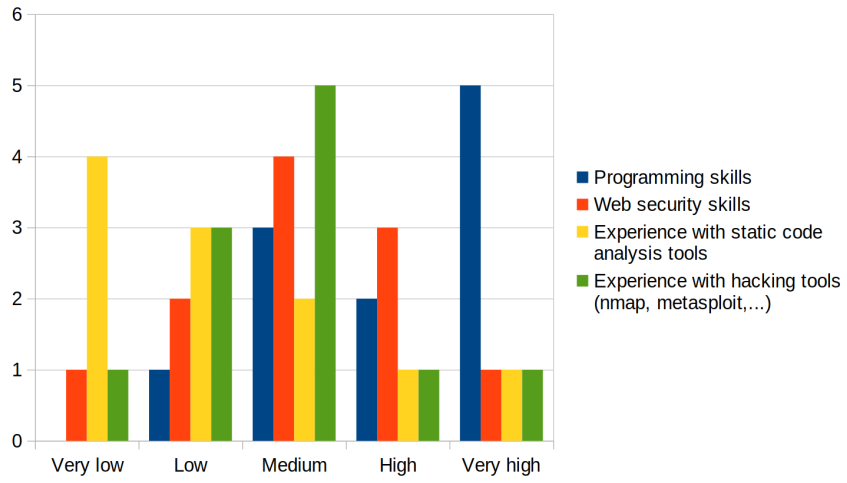


Figure 14.13: Pre-Survey CTF event to check the skill level. (n=11)

are projects [194] that were insecurity-refactored to contain different vulnerabilities. The groups and the corresponding exercises are described in the following sections.

Surveys were used for evaluation of the experiment. At the beginning, a pre-survey was provided to get information about the skill level of attendees. The exercise itself was a bit different for each group. Both used insecurity-refactored projects. For both groups the insecurity-refactored projects were hosted in virtual machines. This allowed to check if the insecurity refactoring actually maintained the external behavior of the programs. After the experiment, a post-survey was provided to see if the exercises were perceived as difficult or realistic and if attendees experienced a skill increase.

Experienced group

The first group was a mix of people with different backgrounds. All of them were training for an upcoming cyber security challenge. Figure 14.13 shows the skill level in different categories. The skill level in programming is overall very high. Also the web security skills are towards medium high rating. The experience with different hacking tools is seen as medium and the experience with static code analysis is low. This group has already experience with capture the flag events from attending other training events. Further, the group is described as the experienced group ("Exp.").

Based on the experience of the group, the idea for this exercise was to provide the attendees with the insecurity-refactored projects in virtual

Project	Type	Input parameter	DF Pattern	Insuff. San.	special	G1	G2	G3	G4
phpBB	SQLi	user.php (style)			hint	✓	✓	✓	✓
phpBB	SQLi	memberlist.php (g)	redirect [man.]						
phpBB	SQLi	posting.php (t)	backdoor int cast [dyn.]						
emonCMS	XSS	compare.php (feedA)	Class storage [sca]	htmlspecialchars		✓	✓	✓	
emonCMS	SQLi	admin_controller.php (perPage)		htmlspecialchars				✓	
phpRedisAdmin	XSS	view.php (page)	backdoor expl/imp. [dyn/sca]		.git dir		✓	✓	
Adminer	XSS	table.inc.php (table)	function call by string [sca]						

Table 14.3: Insecurity-refactored projects for the experienced group (Exp.).

machines like in a CTF event. The attendees were supposed to use their own strategies for detecting vulnerabilities. Attendees had access to the virtual machines that also allowed them to access the source code of the projects.

Table 14.3 shows an overview of the insecurity-refactored projects. The following four projects were used for insecurity refactoring: *phpBB*, *EmonCMS*, *phpRedisAdmin* and *Adminer*. Overall we injected four SQL Injection vulnerabilities and three Cross Site Scripting vulnerabilities spread over the four projects. The phpBB project uses self defined functions for getting user data. We have added a project specific pattern that represents that functions. Without that pattern the Insecurity Refactoring tool would not be possible to detect PIPs in phpBB. Accordingly, the phpBB did not add any PIPs to the initial evaluation based on scanning open source projects. The project specific pattern can be found on GitHub but is disabled at default. Data flow patterns were added, of which three are difficult for static code analysis tools and two difficult for dynamic testing tools. One pattern also used the *function call by string* pattern described earlier. One vulnerability was a plain SQL Injection without any sanitization methods and two vulnerabilities used an insufficient sanitization method for the vulnerability or context. The goal was to create vulnerabilities with different difficulty levels. Additional difficulties can be achieved by adding data flow patterns. Table 14.3 shows these patterns and the corresponding difficulties for different approaches are listed. The difficulties vary for the different approaches to discover a vulnerability. Static code analysis tools (sca), dynamic testing tools (dyn) and manual inspections (man) have different difficulty levels. For example, a dynamic tool has problems to detect backdoors that require specific inputs to bypass sanitization. In contrast, static code analysis tools usually have more problems detecting different dynamic programming approaches or specific source code patterns [207] [208].

Attendees were grouped into four teams who worked together to find the vulnerabilities deployed in the virtual machines. The groups had to provide a report to score points. The report had to contain how the students discovered the vulnerability, how the vulnerability can be exploited and

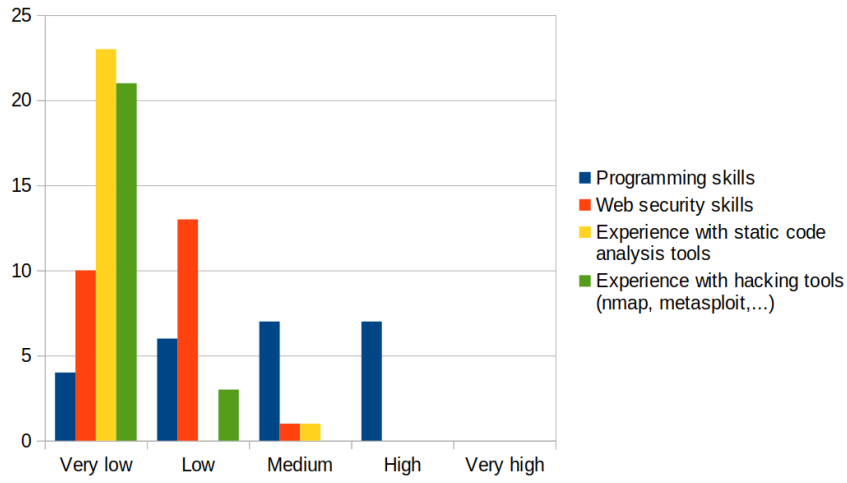


Figure 14.14: Pre-Survey student exercise event to check the skill level. (n=24)

how the vulnerability can be patched. Having a report about discovery, exploitation and patching allowed us to analyze how the teams solved the tasks. The event ran for 24 hours. In the first hours, the groups had to discover the vulnerabilities without any further help. After the initial 10 hours, hints about the vulnerabilities were released. For example, "Some users reported that changing the style of phpBB is buggy." In this case the injected vulnerability used the style parameter in *user.php* for a SQL Injection vulnerability.

Beginners group

A second evaluation as a learning example was done with a group of students. The students were relatively new to software security. Figure 14.14 shows their initial skill levels. It shows that the programming skills are higher than the web security skills. This kind of skill level was expected because the students study computer science and the exercise was done for a software security class. The group will be described as beginners group ("Beg.").

Because the group was not experienced with using any static code analysis or dynamic testing tools, the exercise itself had to be different. Table 14.4 shows the insecurity-refactored projects that were used in this exercise. Only phpBB and EmonCMS were used to injected different vulnerabilities. This time four SQL Injection and two Cross Site Scripting vulnerabilities were injected in the projects. All of them contained different data flow patterns

Project	Type	Input parameter	DF Pattern	Insuff. San.	special
phpBB	SQLi	user.php (style)			Parameter list
phpBB	SQLi	memberlist.php (g)	backdoor int cast [dyn.]		Parameter list
phpBB	SQLi	posting.php (p)	redirect [man.]		Parameter list
emonCMS	XSS	compare.php (feedA)	Deactivated default san. [sca]		Parameter list
emonCMS	SQLi	admin_controller.php (perPage)	function call by string [sca]		Parameter list
emonCMS	XSS	dailyhistogram.php (kwhd)	comparing different types [man]		Parameter list

Table 14.4: Insecurity-refactored projects for the beginner group ("Beg.").

to again make it difficult for static code analysis tools or dynamic testing tools. Except for the very simple SQLi vulnerability in phpBB, no other vulnerabilities are the same as in the data set for the experienced group. Some inputs are the same, but different data flow patterns make them different from each other.

Because the students were not familiar with using static code analysis and dynamic tools, they got tutorials on how to use such tools. Also the students were provided with a static code analysis tool and a dynamic tool that they could use. For the exercise, the students had to scan the provided source code with the static code analysis tool. As the next step, they had to scan the insecurity-refactored projects with the provided dynamic tool. The students got the insecurity-refactored projects deployed in virtual machines and they separately got the corresponding source code. As the last step the students were provided with a list of the vulnerable parameters. This allowed them to check the tools' results and they could manually inspect the remaining undetected vulnerabilities. For each of the steps, the students had to report if it is possible to exploit the discovered vulnerabilities. No patching of the vulnerabilities was required. The time frame for this experiment was four weeks.

Results

First of all, the insecurity-refactored projects with different patterns were deployed in virtual machines. No strange behavior of the projects was reported. Accordingly, the insecurity-refactored projects performed normally as long as no vulnerability was exploited. One problem in the experienced group was revealed that at some point attendees found out that the latest version on GitHub had been used as the base for insecurity refactoring. Then they started to use the *diff* command on the insecurity-refactored projects to find further vulnerabilities. Table 14.3 shows for each group (G1-G4) what vulnerabilities have been reported. Three of seven vulnerabilities were not reported at all. A problem in the beginner group was that it was forced to do the exercise from home. Therefore, they could not be guided well to

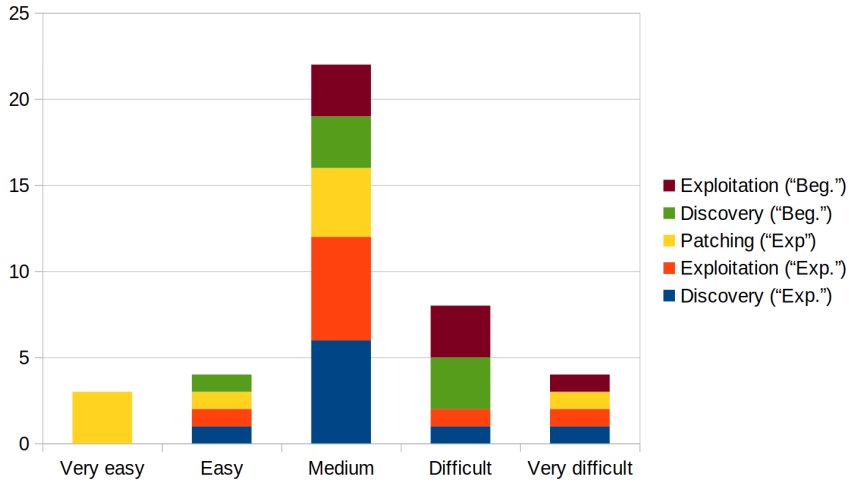


Figure 14.15: Survey results on the difficulty of tasks. (Discovery, Exploitation and Patching) (n=16)

use the given tools and had to rely on the provided tutorials. This was a hurdle that many of the beginners could not overcome and not all of them finished. It was not mandatory to finish the exercise and only 7 of the initial 24 attendees actually finished the exercise.

In the post-survey, attendees were asked how difficult the tasks had been for them. Figure 14.15 shows the results combining the difficulty of discovery, exploitation and patching. For the beginners group the question about patching did not exist because they did not patch the vulnerabilities in their exercise. Overall, the results are towards medium difficulty with being a bit more towards the difficult side. For an exercise, the medium difficulty is optimal. It does not overwhelm learners and is not too easy to solve. Experienced attendees point out that they were a bit overwhelmed by the large projects. Additionally, they pointed out that the tasks were not isolated like in other CTF events. No such complaints were voiced in the beginners group, probably because they got a list of vulnerable parameters. The patching by the experienced group was described as more on the easy side. The reason is that most teams used the master version on GitHub as a patch solution. That is a correct solution, but does not require any skills to patch the vulnerability.

Figure 14.16 shows the results of the questions on how real the tasks were considering bug bounty or code inspections tasks in real life. The results of both groups are shown in the diagram. It indicates that the exercise was

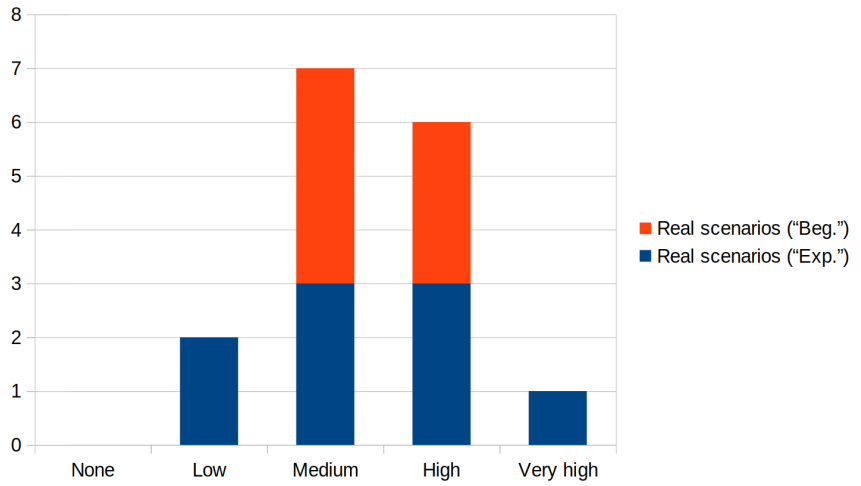


Figure 14.16: Post-survey results about how similar the exercise is to real penetration testing/bug bounty scenarios. (n=16)

close to a real life example. Only two attendees answered that closeness to reality was low.

Attendees were asked how they think their software security skills improved by that event. The results are shown in figure 14.17. First of all, two members of the experienced group did not see any skill increase by the event. The other 14 attendees answered that they experienced a skill increase in software security skills by the event.

14.7.3 Comparative analysis

Previous approaches have used similar procedures to inject vulnerabilities. Table 14.5 shows an overview of methods to inject vulnerabilities from both LAVA and EvilCoder as well as Insecurity Refactoring.

Functional comparison

LAVA uses a dynamic taint analysis to detect DUAs (Dead, Uncomplicated and Available Data). In words, a DUA is a user-controlled input that does not change any control flows and is not concatenated with other variables. Then they search for attack points (ATP) which are near DUAs. An ATP is a sink that can be transformed to create a vulnerability. The vulnerability injection transforms the ATP by adding a conditional usage of the corresponding DUA.

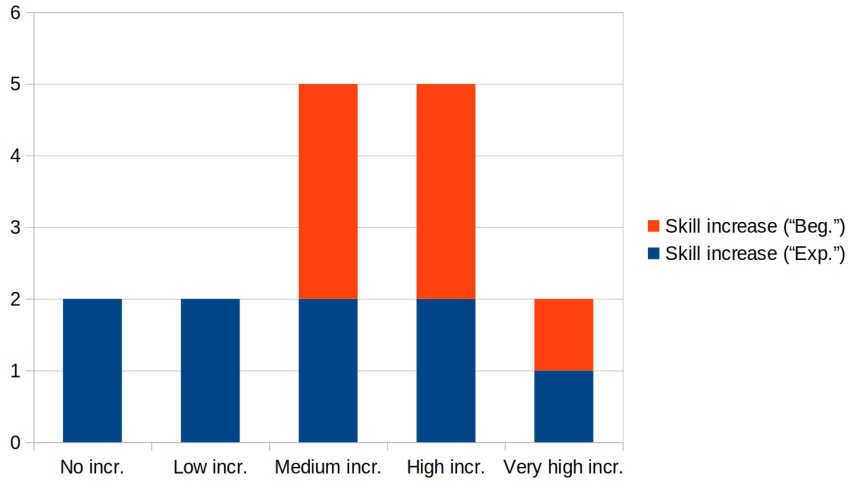


Figure 14.17: Post-survey results if attendees experienced a skill increase from the exercise. (n=16)

Method	LAVA	EvilCoder	Insecurity Refactoring
Language	C/C++	C/C++	PHP
Detection	DUA (dynamic)	Code Property Graph 1. backward 2. forward (CFG)	ACID Tree Context of input
Injection	Use DUA in sink	Invalidate security mechanisms or use security anti patterns	1. Ins. sanitization function 2. Add data flow pattern
Realismn	Synthetic	Artificial	Patterns stem from CVEs

Table 14.5: Comparing methods of LAVA, EvilCoder and Insecurity Refactoring.

Compared to our course of action, the dynamic approach requires a running setup of the program. First of all, that makes the scanning effort more difficult. Nevertheless, the detection should be more precise. Additionally, the authors state that the injected vulnerabilities are synthetic, therefore only exploitable if specific inputs are provided. The condition allows the program to run as intended as long as the specific input is not provided. Therefore, they state that it makes the vulnerabilities more realistic. Nevertheless, the injected vulnerability patterns do not stem from real vulnerabilities.

The EvilCoder approach uses a Code Property Graph. The detection of potential injection locations is done in two steps. In the first step, for all potential sinks a backwards taint analysis on the Code Property Graph is started to find sources. In the second step, for each potential path from a sink to a source, a forward analysis (source to sink) on the Control Flow Graph is started. In the forward analysis, it is searched for security checks that influence the control flow based on data from the tainted variable. These security checks are transformed to inject the vulnerability. It can either be injected by invalidating the security checks or by the use of a security anti-pattern. A security check is invalidated by transforming the conditions to always being true or false. A security anti-pattern transforms the sink to use patterns that are always critical. For example, a `printf("%s", buf)` is replaced by `printf(buf)`.

Compared to our approach, the injected vulnerabilities are artificial. The approach does not ensure the normal behavior of the program afterwards. The injected vulnerability might be triggered all the time. Their approach uses a concept to remove security checks. Many C/C++ vulnerabilities are related to memory bugs that make length checks critical. Our approach can replace security checks and functions that transform data (e.g. *htmlspecialchars()*). The approach to add anti-patterns is similar to our approach of replacing secure sinks with insecure sinks.

Overall, our approach is focused towards PHP and corresponding typical vulnerabilities. As PHP is typically used in web-based applications, the vulnerabilities heavily depend on the context. The ACID tree is another analysis model on top of the Code Property Graph that allows to analyze the context of given user input. This gives us the opportunity to be more specific whether a sanitization function is sufficient or not. The other approaches do not consider the context. LAVA tries to minimize that problem by using variables that are not concatenated with other variables. EvilCoder instead invalidates the whole security check independent of the context. In contrast, our approach is precise, which has the disadvantage of not finding as many potential injection paths. But it maintains the normal usage of the program.

Our approach provides a PL/V pattern language that allows to describe the critical patterns. Additionally, our patterns stem from existing CVEs to maintain patterns of realistic vulnerabilities. By definition all injected vulnerabilities are artificial, including our approach. Insecurity Refactoring injects patterns that stem from CVE reports in existing projects to keep the vulnerabilities as realistic as possible. In addition, the data flow patterns can be used to introduce difficulties based on the pattern.

Experimental comparison

The EvilCoder approach to find PIPs is similar to our approach. An experiment with the same programs as input is not possible because EvilCoder uses C and our approach uses PHP as input. Nevertheless, we compare their results from scanning open source projects to our results from scanning open source projects in detail. Table 14.6 shows the results that EvilCoder got on four open source projects and that we achieved for the same number of projects. The results include a special pattern for the custom *phpBB* function to retrieve user data. Without that pattern, Insecurity Refactoring cannot find a PIP. The results without the pattern are shown in brackets. First of all, the results show that Insecurity Refactoring finds a lot more sources and sinks compared to EvilCoder. A reason for that is that PHP web vulnerabilities have a different kind of sinks and sources. For example, for XSS every function that prints text on a web page will be a possible sink. This includes functions that just print static text. A unique source-sink stands for at least one data flow path between a specific source and sink. The Insecurity Refactoring flags an ACID tree that contains at least one path from source to sink as a PIP. It does not count each leave as an additional PIP. In contrast, source-sink paths count all possible paths that are found between sources and sinks. First of all, the results show that EvilCoder finds more unique source-sinks per given sources and sinks compared to Insecurity Refactoring. Compared to the lines of code, the unique source-sink pairs found are in a similar range. For the source-sink paths, EvilCoder finds more paths. The different code base and vulnerabilities might explain that. Nevertheless, an implementation difference here is that EvilCoder tracks each control flow path that can be taken. The ACID tree combines such control flow paths. Another path (split in the ACID tree) would only be created when an *if* statement contains a union that then will be represented as an excluding ($\oplus$) split.

As a next step, the vulnerability injection can be evaluated. EvilCoder ships only two kinds of instrumentation on the GitHub project. They state that it can be extended to create more variations. Here is a gap between

EvilCoder				
	libpng	vsftpd	wget	busybox
Lines of code	40,004	20,046	137,234	265,887
Sources	9	3	21	152
Sinks	98	13	453	573
Unique Source-Sink	158	22	22	30
Source-Sink paths	22,516	786	1,882	2,905
Insecurity Refactoring				
	Adminer	EmonCMS	phpBB	phpRedisAdmin
Lines of code	27,606	26,383	289,800	2,022
Sources	752	138	552 (223)	210
Sinks	1,386	3,417	3,795 (3,795)	478
Unique Source-Sink	39	13	188 (0)	25
Source-Sink paths (PIPs)	65	14	292 (0)	30

Table 14.6: Comparing results of EvilCoder and Insecurity Refactoring.

their approach and ours. Our approach evaluates if a sanitization function is sufficient for a given context. EvilCoder only provides the possibility to replace an *if* statement with an instrumentation. Our approach allows more variations for a given PIP (source-sink). For example, for a given PIP it is allowed to replace a source with 5 other sources, 10 different data flow patterns can be inserted, and 9 different sanitization functions would be insufficient. This allows to inject a vulnerability in $5 * 10 * 9 = 450$ different permutations. This is an advantage of our approach over the EvilCoder approach. Small patterns can be defined and those patterns can be combined to inject vulnerabilities.

Nevertheless, the comparison between two tools that work with different vulnerabilities and on source code in different programming languages cannot be compared empirically. Our experiment shows that EvilCoder provides more possibilities to inject vulnerabilities that use different data flow paths. In contrast, the Insecurity Refactoring approach allows to inject many different permutations of a vulnerability.

14.8 Discussion

Insecurity refactoring is a novel method that injects vulnerabilities in projects based on source code patterns gathered from vulnerabilities in CVE reports. It shows that PIPs can be found and transformed into vulnerabilities. Also some patterns can be added that make detection by static code analysis tools difficult. One ethical question is if developing and publishing such

a tool might be more harmful than useful. The main idea is to actually use insecurity refactoring to create learning examples. The tool could also be used maliciously to inject vulnerabilities in projects that are actually deployed in productive systems. For example, a malicious *Git* software could use insecurity refactoring to inject vulnerabilities before it pushes code changes to the *Git* server. Such an attack scenario requires that the *Git* server does not review pull requests. Another scenario might be that the *Git* client that pulls the source code is malicious. The client could perform insecurity refactoring on each pull request. This is a possible attack scenario but requires to add the malicious *Git* client on the server in the first place. We see such attack scenarios as more artificial than actually relevant in practice.

As learning examples, a defined difficulty of the vulnerabilities would be beneficial. Our evaluation shows that most of the attendees reported skill increase attending an event that used vulnerabilities generated by insecurity refactoring. Some form of difficulty rating for the different patterns would be useful. For now we can only predict the difficulty based on how large the initial project source code is and whether we added some special patterns. The results of the evaluation show that some attendees would like to have hints as to where vulnerabilities are. Accordingly, the scenario of the exercise itself is also important. Insecurity refactoring allows to inject vulnerabilities in real projects to get vulnerability examples as real as possible. Nevertheless, the exercises in which the insecurity-refactored projects are used may be very different. The results show in two different scenarios that the insecurity-refactored projects can be used as learning examples.

The difficulty varies based on what kind of learning example the insecurity-refactored source code is used for. If it is used to teach the use of static code analysis tools, a vulnerability without a special difficult static code analysis pattern is not difficult. But it might be difficult if the vulnerability has to be found manually. In the end, the difficulty of the insecurity-refactored vulnerabilities heavily depends on the task.

Our evaluation on open source projects showed a problem in finding control functions using the *Ctrl* function. This is a classical NP-hard problem because all possible paths from one statement to another statement have to be created and each of these paths has to be checked if it contains any sanitization check methods. Most of the time, possible paths are short and the query runs fast. Nevertheless, some projects contain so many possible paths (high complexity) that the query run time increases to an inconveniently long period (>2 minutes) on modern hardware. As a solution we scanned in two steps. The first step ignores any control function checks. If the first

step finds a PIP or vulnerability, a second analysis is done using the control function checks.

Another problem is that the control property graph does not support object-oriented data flow. Method calls from objects are resolved correctly, but data that is stored in object variables is not tracked. This decreases the chance to find PIPs. Especially for SQLi, many database drivers are stored in objects (db wrapper) or the queries are constructed using data-represented objects [205].

The evaluation shows positive results for a small survey size ($n = 9 + 7 = 16$). The results of the evaluation as a code inspection task shows that the small test group had skill improvements. The small survey size cannot be used to statistically proof that the insecurity refactored projects are always beneficial as learning examples. At least for that small test group it showed skill improvements. Future work should use larger test groups ($n > 100$) to statistically verify the initial results of the small sample size. However, the usefulness of software security exercises not only depends on the vulnerability itself. The results show that insecurity-refactored vulnerabilities are usable for software security exercises. One problem of such exercises is the time it takes to create vulnerabilities in a real scenario. The results show that insecurity refactoring can inject vulnerabilities with different patterns into existing projects. Accordingly, the scenario where vulnerabilities appear is as real as possible. Overall, the concept of insecurity refactoring works on open source projects without violating the definition of insecurity refactoring (not changing the external behavior in normal usage).

14.9 Conclusion

Our approach for insecurity refactoring shows that vulnerabilities can be injected into open source projects by using static code analysis approaches. The ACID tree is introduced as an analysis model for finding PIPs and vulnerabilities. Finding locations of PIPs has the same limitations as finding vulnerabilities in the first place. A precise approach was used to mitigate any false positive results where injected vulnerabilities would have a high chance of not being exploitable. A false positive PIP might break the normal use of the project, hence breaking the insecurity refactoring definition. The PIP can be simple. The injected vulnerability can be made difficult by adding data flow patterns. These patterns can be so difficult that the ACID tree approach cannot detect them anymore. Accordingly, the injection of vulnerabilities can be a lot easier than the detection of the injected vulnerabilities. If any useful attack scenarios of insecurity refactoring are found, the automated

detection of these vulnerabilities will be more difficult.

The focus of insecurity refactoring is to inject vulnerabilities with different source code patterns. The PL/V pattern language allows to define the source code patterns in an independent language. To extend the tool to support other programming languages only the language patterns have to be rewritten and the Code Property Graph has to be created for that language. A first evaluation shows on a small sample size that insecurity refactoring can be used to teach software security skills. Compared to other approaches our focus relies on to create vulnerabilities realistic as possible. The approach shows that the concept works with different source code patterns. The different patterns also allow to create many permutations of vulnerabilities. This enables repeatedly using insecurity refactoring to teach software security skills.

References

[183] A. Alhuzali, R. Gjomemo, B. Eshete, and V. N. Venkatakrishnan. NAVEX: Precise and scalable exploit generation for dynamic web applications. *Proceedings of the 27th USENIX Security Symposium*, pages 377–392, 2018.

[184] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and Flexible Discovery of PHP Application Vulnerabilities. *Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017*, pages 334–349, 2017. doi: 10.1109/EuroSP.2017.14.

[185] T. Boland and P. E. Black. Juliet 1.1 C/C++ and Java Test Suite. *Computer*, 45(10):88–90, oct 2012. ISSN 1558-0814. doi: 10.1109/MC.2012.345.

[186] J. Burket, P. Chapman, T. Becker, C. Ganas, and D. Brumley. Automatic problem generation for $Capture - the - Flag$ competitions. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*, 2015.

[187] P. Chapman, J. Burket, and D. Brumley. {PicoCTF}: A {Game-Based} computer security competition for high school students. In *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*, 2014.

[188] N. Deo. *Graph Theory with Applications to Engineering and Computer Science (Prentice Hall Series in Automatic Computation)*. Prentice-Hall, Inc., USA, 1974. ISBN 0133634736.

[189] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. LAVA: Large-Scale Automated Vulnerability Addition. *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, pages 110–121, 2016. doi: 10.1109/SP.2016.15.

[190] W. Du. SEED: Hands-on lab exercises for computer security education. *IEEE Security and Privacy*, 9(5):70–73, 2011. ISSN 15407993. doi: 10.1109/MSP.2011.139.

[191] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Professional, 1999. ISBN 0201485672.

[192] Github. `https://github.com/`, 2022.

[193] Insecurity Refactoring. `https://github.com/fschuckert/insecurity-refactoring`, 2022.

[194] Insecurity Refactoring code samples. `https://github.com/fschuckert/insec_samples`, 2022.

[195] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243804.

[196] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a Program Query Language. *ACM SIGPLAN Notices*, 40(10):365, 2005. ISSN 03621340. doi: 10.1145/1094811.1094840.

[197] K. Maruyama and T. Omori. A Security-Aware Refactoring Tool for Java Programs. *Proceedings - International Conference on Software Engineering*, pages 22–28, 2011. ISSN 02705257. doi: 10.1145/1984732.1984737.

[198] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004. doi: 10.1109/tse.2004.1265817.

[199] W. F. Opdyke. *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign, 1992.

[200] J. Pewny and T. Holz. Evilcoder: Automated bug injection. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 214–225, 2016.

[201] PHP Documentation. `https://www.php.net/manual/`, 2021.

[202] PHP repository - backdoor commit. `https://github.com/php/php-src/commit/c730aa26bd52829a49f2ad284b181b7e82a68d7d`, 2021.

[203] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017. doi: 10.14722/ndss.2017.23404.

[204] Z. C. Schreuders, T. Shaw, M. Shan-A-Khuda, G. Ravichandran, J. Keighley, and M. Ordean. Security scenario generator (SecGen): A framework for generating randomly vulnerable rich-scenario VMs for learning computer security and hosting CTF events. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*, Vancouver, BC, Aug. 2017. USENIX Association.

[205] F. Schuckert, B. Katt, and H. Langweg. Source Code Patterns of SQL Injection Vulnerabilities. *International Conference on Availability, Reliability and Security*, 2017. doi: 10.1145/3098954.3103173.

[206] F. Schuckert, M. Hildner, B. Katt, and H. Langweg. Source Code Patterns of Buffer Overflow Vulnerabilities in Firefox. *Proceedings of Sicherheit 2018*, pages 107–118, 2018. doi: 10.18420/sicherheit2018_08.

[207] F. Schuckert, B. Katt, and H. Langweg. Difficult XSS code patterns for static code analysis tools. In *Computer Security - ESORICS 2019 International Workshops, IOSec, MSTEC, and FINSEC, Luxembourg City, Luxembourg, September 26-27, 2019, Revised Selected Papers*, volume 11981 of *Lecture Notes in Computer Science*, pages 123–139. Springer, 2019. doi: 10.1007/978-3-030-42051-2_9.

[208] F. Schuckert, B. Katt, and H. Langweg. Difficult SQLi Code Patterns for Static Code Analysis Tools. *Norsk IKT-konferanse for forskning og utdanning – NISK Norsk informasjonssikkerhetskonferanse*, 2020(3), 2020. doi: https://ojs.bibsys.no/index.php/NIK/article/view/892.

[209] B. Stivalet and E. Fong. Large Scale Generation of Complex and Faulty PHP Test Cases. *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, pages 409–415, 2016. doi: 10.1109/ICST.2016.43.

[210] S. Thomas, L. Williams, and T. Xie. On automated prepared statement generation to remove SQL injection vulnerabilities. *Information and Software Technology*, 51(3):589–598, 2009. ISSN 09505849. doi: 10.1016/j.infsof.2008.08.002.

[211] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. *Proceedings - IEEE Symposium on Security and Privacy*, pages 590–604, 2014. ISSN 10816011. doi: 10.1109/SP.2014.44.

[212] M. M. Yamin and B. Katt. Use of cyber attack and defense agents in cyber ranges: A case study. *Computers & Security*, 122:102892, 2022. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2022.102892.

[213] M. M. Yamin and B. Katt. Modeling and executing cyber security exercise scenarios in cyber ranges. *Computers and Security*, 116:102635, 2022. ISSN 01674048. doi: 10.1016/j.cose.2022.102635.

Chapter 15

Systematic Generation of XSS and SQLi Vulnerabilities in PHP as Test Cases for Static Code Analysis

15.1 Introduction

Static code analysis tools are a good approach to find software vulnerabilities early in the development phase. False positive (FP) and false negative (FN) reports are problematic. A false negative report will probably pass the vulnerability to a productive system where the vulnerability can be harmful. In contrast, false positive reports take time of developers to review them. If the false positive reports are too many developers might overlook true positive reports. The research about *soundness* of static code analysis tools tries to make the tools more precise and meaningful (e.g. [227], [236]). "Sound means every finding is correct. The tool need not produce a finding for every site; that is completeness." [216]

We provide a tool that can generate synthetic test cases. Synthetic test cases are usually small test cases that contain only the vulnerability. In contrast, a not synthetic test case would be by using source code from an open source project that contains a known vulnerability. The generated synthetic test cases use the combination of source, sanitization, context and sink to decide if the test cases are vulnerable or not vulnerable. The combination of all the different parts allow to identify the problems of static code analysis tools. This makes the generated test cases relevant to test static code analysis

tools.

15.1.1 Related Work

Anand [220] provides an overview of automated test case generation based on the tested source code. It is split into different types of automated test case generations. The different types look into different parts like the inputs/outputs of a program or the internal structure. The type *program-based* test case generation uses symbolic execution to define testing inputs that reach specific symbolic execution paths. It is used to generate test data to improve code coverage and expose software bugs (e.g. [222]). The *model-based* test case generation sees the software to be tested as a black box. At first it defines abstract test suites. For each tested software a concrete test suite has to be built based on the abstract test suite. The *data-centric* test case generation uses a list of inputs and the corresponding expected outputs to test the software. This allows to automate the testing process. The combinatorial interaction testing (CIT) has different inputs and for each input there is a list of possible inputs. A combination of these inputs is then used to test the software. Nie and Leung [228] provide a survey of CIT. The *search-based* test generation searches for good test cases. The goal is to maximize the achievement of test goals and to minimize the testing cost. A search based optimization algorithm guided by a fitness function is used to find good test cases. For example, Harman and McMinn [226] used structural testing goals to achieve a *search-based* test case generation. Our approach can be classified as a *data-centric* test case generation. The resulting source code can be seen as multiple inputs where each pattern is an input. All the combinations of patterns result in different test cases for static code analysis tools.

Test suites that are used to evaluate static code analysis tools are usually synthetic. The Juliet [221] test suite has been generated by the NSA Center for Assured Software. The Java test suite contains 28,881 test cases under 112 different CWEs. The C/C++ test suite contains 64,099 test cases under 118 different CWEs. A relatively new C# test suite contains 28,942 test cases under 105 different CWEs. Each test case targets only one flaw, but some test cases also contain multiple flaws. Common additional flaws are hardcoded passwords or unreachable code.

The STONESOUP test suite [223] is split into three phases. The first and second phases contain small test cases. The third phase contains large test cases with injection vulnerabilities. 16 open source projects are used as a base and different vulnerabilities are injected. Specific inputs trigger the vulnerabilities. The test suite of phase three contains 4,582 test cases in C

and 3,188 test cases in Java. There is more research about automatically injecting vulnerabilities in source code (e.g [224], [229] [234]).

Hao [219] constructed a test suite based on known vulnerabilities in open source projects. The projects are reduced to only contain the *original* vulnerability. That vulnerability is transformed into a *base* vulnerability that only contains the sink and source. The *base* vulnerability allows to check if the static code analysis tools can detect it. *Features* are added to the base vulnerability to create variants of the vulnerability. The concept of a *base* vulnerability and *features* allows to verify what features the static code analysis tool can support.

Stivalet and Fong [235] developed a tool to generate many synthetic test cases in PHP. The current test suite contains 42,212 test cases with different CWE categories. Our approach is similar and our test suite is an addition to the existing PHP test suite. A similar approach is used and extended by decision trees and additional patterns (*context*, *dataflow*). This makes our SQL injection and XSS test cases more detailed than the test cases in these categories by Stivalet and Fong [235].

15.2 Methodology

Our methodology is to generate test suites for static code analysis tools. The generation process is based on source code patterns that are described in section 15.3. The resulting test suites are scanned with two commercial static code analysis tools. Static code analysis metrics are calculated based on the results. Additionally, the results are analyzed in to pin point problems of the static code analysis tools. The static code analysis tools are seen as black boxes because commercial tools do not provide access to the source code. The results show if the test suites are usable as a benchmark for static code analysis tools. Finally, expert interviews are used to evaluate if the test suite generation is useful or if it has any flaws.

15.3 Test case generation

We have developed a framework [214] that can either inject vulnerabilities in a PHP project or can generate test cases. This section describes how the test cases are generated.

15.3.1 Design of Test Cases

The design of the test cases is similar to test cases by Stivalet and Fong [235] that have been used to create the PHP Vulnerability Test Suite. A combination of different source code parts is used to generate a test case. Based on the combination, it either is a vulnerable test case or it is not vulnerable. A vulnerable test case means that the source code contains a vulnerability. The vulnerability might not be exploitable, but the likelihood of being exploitable is high. A not vulnerable test case means there is no possibility to inject critical characters. Accordingly, the test case cannot be exploited. The decision of being a vulnerable or not vulnerable test case will be explained in this section. We have the focus on Cross Site Scripting (XSS) and SQL injection (SQLi) and extend the PHP Vulnerability test Suite. For both vulnerability categories (XSS and SQLI) the test cases from our framework cover all the test cases of the PHP Vulnerability Test Suite in these categories. It extends these test cases by adding more sanitization methods and by distinguishing between the context and the sink.

15.3.2 Internal Structure

The internal structure to generate the test cases is based on five pattern types:

- *Source* patterns provide user data. Depending on the source, the user provided data is filtered (e.g. only pass integer value) or the source provides unfiltered data.
- *Sanitization* patterns are functions that filter or encode characters. This category includes functions that transform data (e.g hash function, string manipulations). Some patterns do not filter any characters.
- *Dataflow* patterns are a list of code statements that pass data from an input to an output without transforming the data. For example, a combination of functions *explode* and *implode* can be used to transform data into an array and back to the original form. A simple assignment is also a dataflow pattern.
- *Context* patterns describe the context of the user input that will be passed to the sink: For example, if the user input is put inside quotes or apostrophes. A *context* pattern is always bound to a specific sink pattern category. For example, an SQL statement is dynamically constructed by using user input. This *context* pattern can only be used for sink patterns that are from the SQL injection category.

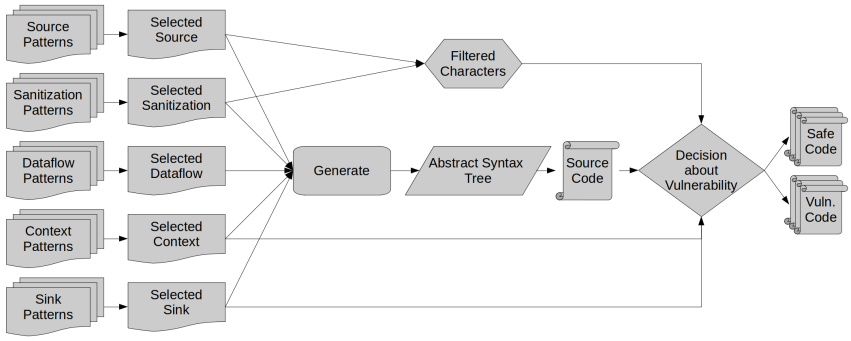


Figure 15.1: Overview of generating the test cases.

- *Sink* patterns are functions where sensitive operations are executed. If unfiltered user data reaches such functions, it will be a vulnerability.

Figure 15.1 shows the internal structure to generate the test cases. For each test case a source, sanitization, dataflow, context and sink pattern is selected. The generation chains the patterns together and generates an Abstract Syntax Tree that is used to generate the final source code. A decision tree is used to decide if a test case is vulnerable or not vulnerable (safe).

15.3.3 Pattern for test case generation

The patterns are stored in *JSON* files. Table 15.1 shows the main attributes of the patterns. The type defines the programming construct that is required for the input, output and the pattern itself. The data attribute describes the data types that can be used as input and the resulting output data type. For example, a pattern (*data (input): Numeric, data (output): String*) requires a numeric input and the output has the data type *String* that contains the input.

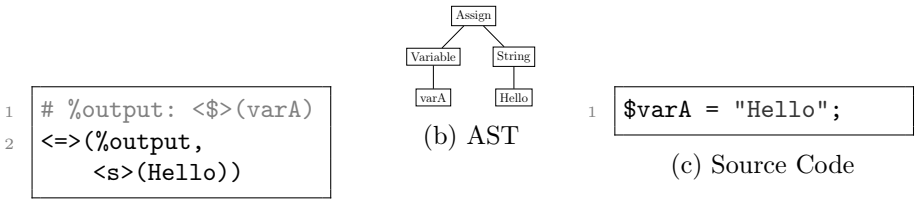
The source code representation of each pattern is stored in the PL/V language. The PL/V is a context free language that represents an Abstract Syntax Tree in text form. For example, figure 15.2 shows the PL/V pattern

$$<=> (\%output, < s > (Hello))$$

that represents an assignment (<=>) with the string value ($< s >$) *Hello*. The operand is written like $< operand >$ and the operators are listed inside brackets. The symbol % is used for variables. The Special variables $\%input$ and $\%output$ are used to chain the patterns together. For each pattern the

Table 15.1: Attribute descriptions of the patterns.

Key	Description	Pattern
name	A unique name for the pattern	all
type	The type of the pattern (Expression or Statement)	all
type (input)	The type of the input pattern (Expression, Statement, Variable)	all except source
type (output)	The type of the output pattern (Expression, Statement, Variable)	all except sink
data (input)	Data types that can be used as input (Array, String, Numeric, Any)	all except source
data (output)	Data types that can be used as output (Array, String, Numeric, Any)	all except sink
Filtering	Characters that are filtered or escaped	Source, Sanitization
Escapes	Characters that can be used to escape critical characters	Sink
Init	A list of statements that are required for the initialization. Defined in PL/V language.	all
Code	The code of the pattern described in PL/V language	all



(a) PL/V (b) AST (c) Source Code

Figure 15.2: The generation process from PL/V to source code.

Abstract Syntax Tree is generated and then put together into one Abstract Syntax Tree. The test case generation uses variables in between each pattern to pass the variable to the corresponding input and output.

15.3.4 Vulnerability Decision

Each test case can either be vulnerable or not vulnerable. A vulnerability is defined as: "Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source." [215] Our approach uses decision trees that are based on what characters are filtered/escaped, the context and the given sink.

The *escapable* decision is based on what enclosure is used in the *context* pattern. For example, if the enclosure uses the apostrophe character, it will be checked if the test case filters apostrophe characters. If the sink allows escaping, it will be checked if apostrophe characters are escaped with the corresponding escape character. If the character is properly escaped or filtered, the test case will not be vulnerable.

The *special chars* decision checks if any special characters are not filtered or properly escaped. Special characters in this case are all characters (including white spaces) that are not alphabetic or numeric. If user input reaches the decision *special chars*, it is a critical part. Accordingly, only uncritical characters are allowed.

Figure 15.3 shows the decision tree for a SQL injection test case. A SQL statement only has either the context of being inside an enclosure or being a plain SQL statement. Accordingly, it simply checks in the plain context if any special characters are allowed. For example, if a SQL statement checks for a personal identification number (PIN), no enclosure will be required because the PIN is numeric. If the input can only be numeric, the test case will not be vulnerable. If special characters are allowed, it could be exploited. In contrast, if an enclosure is used like the apostrophe character, it would be sufficient to filter out any apostrophe characters. If the apostrophe is not

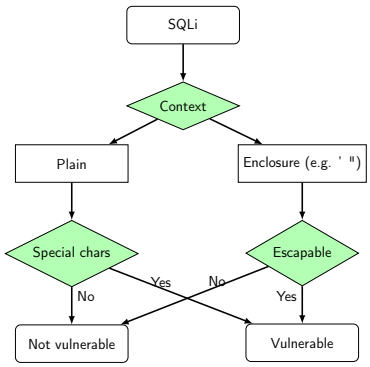


Figure 15.3: Decision tree vulnerable/safe for SQL injection.

filtered out, the enclosure can be escaped that is not intended by a developer.

Figure 15.4 shows the decision tree for a Cross Site Scripting test case. For Cross Site Scripting the context and enclosure can be nested. The context is split into *plain*, *HTML Attribute* and *Javascript*. In a *plain* context, an enclosure does not prevent any injection approaches. Accordingly, it will be checked if any special characters are allowed. Inside *Javascript* context, the decision is the same as for SQLi test cases. Based on an enclosure or not it will be checked for *escapable* or *special chars*. The *HTML Attribute* context is a bit different. Some attributes like *onclick* or *href* are critical because they can be used to execute Javascript code. Accordingly, if such a *JS critical* attribute is the context, it will be checked for *special chars*. If it is no *JS critical* attribute, it will be checked if the attribute uses an enclosure or not. Accordingly, for non *JS critical* attributes the vulnerability decision is the same as for SQL injections.

15.3.5 Code generation

Each code sample uses a *source*, a *sanitization*, a *dataflow*, a *context* and a *sink* pattern to generate the source code. Figure 15.5 shows a generated test case. At the beginning, the database object *$db* is initialized. The source uses a PHP function that filters critical characters. The *sanitization* pattern is a data type cast to integer. The *dataflow* pattern stores the data in an environment variable and reads the stored data from it. The context is a plain SQL statement (Context: *plain*). The sink is a critical SQL function to run the SQL statement. For testing, output statements are added. The test case is not vulnerable because the *sanitization* pattern filters pass only numeric inputs.

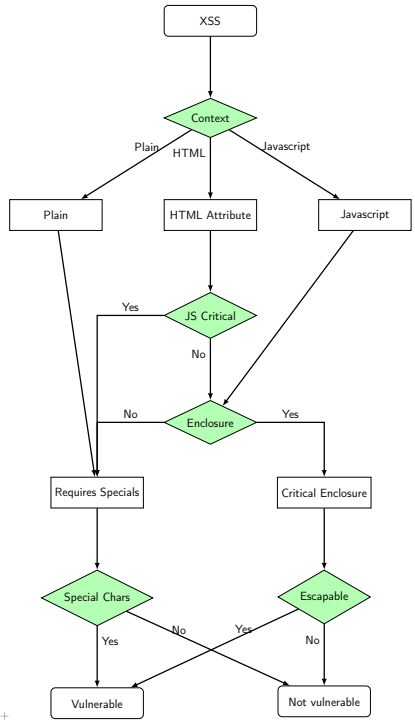


Figure 15.4: Decision tree vulnerable/safe for Cross Site Scripting.

```
1  <?php
2       # Init                                                    Init
3       $servername = "mysql";
4       $username = "username";
5       $password = "password";
6       $dbName = "myDB";
7       $db = new mysqli($servername, $username, $password, $dbName);
8
9       # Sample                                                  Source
10      $tainted = filter_input_array(INPUT_GET,
11                      ["t" => FILTER_SANITIZE_FULL_SPECIAL_CHARS]);
12      $tainted = $tainted["t"];
13
14                                                              Sanitization
15      $sanitized = (int)($tainted);
16
17                                                              Dataflow
18      putenv(("IMPORTANT_VARIABLE=" . $sanitized));
19      $dataflow = getenv("IMPORTANT_VARIABLE");
20
21                                                              Context
22      $context = (("SELECT * FROM users WHERE pin ="
23                      . $dataflow) . ";");
24
25                                                              Sink
26      $db→real_query($context);
27      $results = $db->use_result();                           Sink (output)
28      while(($row = $results->fetch_row()))
29      {
30        echo(htmlentities(print_r($row, true)));
31      }
32 ?>
```

Figure 15.5: Generated Test Case (Not vulnerable)

15.3.6 Tool usage

The tool uses command line instructions to generate the different test suites. The following command has been used to generate the soundness test suite:

```
$ sh run_insec.sh -genFiles
  -generate_samples [target_directory]
  -onlyPattern dataflow:assignment
  -split_by source
```

The commands *genFiles* and *generate_ samples* define that a test suite will be generated in the target directory. The *onlyPattern* parameter defines that only the *assignment* pattern from the *dataflow* pattern category is used. The *split_ by* defines which pattern is used to split the test cases into directories. For the data flow test suite the following command has been used:

```
$ sh run_insec.sh -genFiles
  -generate_samples [target directory]
  -onlyPattern sanitize:nosanitization
  -onlyPattern context:xss_plain
  -onlyPattern sink:echo_func
  -onlyPattern source:_GET
  -split_by source
```

It restricts each test case to be a cross site scripting vulnerability without any sanitizations. The only permutations are found in the *dataflow* patterns.

15.4 Evaluation

It is not within the scope of this paper to evaluate static code analysis tools. For the evaluation, we use two commercial static code analysis tools to see if the generated test suite is useful. These tools were selected after an evaluation of open source and commercial static code analysis tools that support PHP, detect XSS and SQLi and are up to date. Nevertheless, it is not important to evaluate the tools instead the evaluation should check, if the test suites can be used as benchmark and to identify problems of static code analysis tools. Additionally, an expert interview with SAMATE explains the usefulness and why synthetic test cases can be critical to evaluate static code analysis tools.

15.4.1 Generated Test Cases

Table 15.2 shows the patterns that are used to generate the test suites. The patterns stem from previous work [230, 231] on how vulnerabilities occurred in open source projects. Even some patterns stem from evaluating static code analysis tools [232] [233]. Additionally, the PHP documentation [217]

Table 15.2: Patterns that are used to generate the test suites.

Pattern	**Count**
Source	22
Sanitization	117
Dataflow	20
Context (Total)	10
Context (XSS)	7
Context (SQLi)	3
Sink (Total)	23
Sink (XSS)	9
Sink (SQLi)	14

has been reviewed to cover the different SQL database implementations and basic PHP functionality. The sanitization patterns contain functions that do not filter characters. For example, the *strtoupper* function is used to convert a variable to upper case. Also encryption functions are added as sanitization functions because they filter critical characters. As final step, patterns related to Cross Site Scripting and SQL injection from Stivalet and Fong [235] have been added to the patterns.

We decided to generate a soundness test suite and a dataflow test suite. The soundness test suite contains all possible mutations between *source*, *sanitization*, *context* and *sink* patterns. All of these permutations are relevant because the outcome of being vulnerable or not is based on these patterns. The *dataflow* pattern has no influence on being vulnerable or not. The soundness test suite uses only an assignment as *dataflow* pattern. Table 15.3 shows how many test cases have been generated. Not all permutations of the patterns are possible because of different constrains. For example, the input and output data type of the patterns have to match. That leaves 258,432 permutations of the patterns.

The second test suite is the dataflow test suite. Each test case is a simple Cross Site Scripting vulnerability that just uses different *dataflow* patterns. All test cases are vulnerable. The test case that uses an assignment as *dataflow* pattern can be used as evaluation. If the tool detects that test case, the tested static code analysis tool will be able to detect the base test case. Then, if the tool does not detect other test cases in the test suite, the dataflow pattern will be problematic for the tool.

Table 15.3: Test Suite to check proper sanitization (Dataflow pattern:assignment).

Test Cases	**Count**
XSS - Vuln.	23499
XSS - Safe	131481
XSS - Total	154980
SQLi - Vuln	20187
SQLi - Safe	83265
SQLi - Total	103452
Vuln	43686
Not Vuln.	214746
Total Test Cases	**258432**

15.4.2 Static code analysis tools

This sections uses the scan results to show that the test suites can be used to find out what the problems of the static code analysis tools are. We used two commercial static code analysis tools to see if the generated test suites are useful to evaluate static code analysis tools. For the soundness test suite, the following metrics have been calculated:

- Precision: $P = \frac{TP}{TP+FP}$
- Accuracy: $Acc = \frac{TN+TP}{TN+FN+FP+TP}$
- Recall: $R = \frac{TP}{FN+TP}$
- False alarm rate: $F = \frac{FP}{TN+FP}$
- G-Score: $G = \frac{2R(1-F)}{R+1-F}$

Different metrics can be used to compare static code analysis tools. We decided to use the G-Score as a metric because it makes the results comparable in a single metric. The G-Score is a harmonic mean of R_i and $1 - F_i$ that integrates the recall R_i and false alarm rate F_i into one metric [225]. Higher values are better.

Table 15.4 shows the scan results on the soundness test suite. Based on these results, *Tool A* performs better on the soundness test suite. Overall, the tools do not perform well on the soundness test suite.

Our first step to pin point the problems was to check, if the tool do consider the context of each test case. The results are analyzed if the tools

Table 15.4: Scan result on the soundness test suite.

Metric	**Tool A**	**Tool B**
False Positive (FP)	55969	7401
False Negative (FN)	30079	41880
True Positive (TP)	13607	1806
True Negative (TN)	158777	207345
Precision	19.6%	19.7%
Accuracy	66.7%	80.9%
Recall	31.2%	4.1%
False alarm rate	26.1%	3.5%
G-Score	**43.8%**	**7.9%**

Table 15.5: Scan result on the soundness test (plain context).

Metric	**Tool A**		**Tool B**	
False Positive (FP)	4157	(-92.6%)	434	(-94.1%)
False Negative (FN)	9137	(-69.6%)	12860	(-69.3%)
True Positive (TP)	11803	(-13.3%)	1711	(-5.3%)
True Negative (TN)	31527	(-80.1%)	41619	(-79.9%)
Precision	74.0%	(+54.4)	79.8%	(+60.1)
Accuracy	76.5%	(+9.8)	76.5%	(-4.4)
Recall	56.4%	(+25.2)	11.7%	(+7.6)
False alarm rate	11.7%	(-14.4)	1.0%	(-2.5)
G-Score	**68.8%**	(+25)	**21.0%**	(+13.1)

have any different reports if only the context is changed. Based on the results, both tools do not consider the context for XSS or SQLi test cases. Table 15.5 shows the metrics calculated on the test cases with plain context. Plain context means that there is no enclosure. Accordingly, sanitization is not sufficient if it prevents only escaping the enclosure. The metrics show that the tools do perform better without different context.

Additionally, the test suite can be used to see if the tools do not know any sinks (**unknown sinks**) or sources (**unknown sources**). To find missing sinks or source, all vulnerable test cases are checked if the tools reported a vulnerability. If all test cases for a specific sink or source do not have any vulnerable report, the tool does not know that sink or source. There is a possibility that the tool considered that source or sink as safe to use. Black box testing does not allow to distinguish if the tools sees the source/sinks as safe or unknown. Nevertheless, the test cases are vulnerable and the tools

should not ignore the corresponding sources or sinks.

Missing sufficient sanitization is a check, to see if the tool misses any sanitization functions that are sufficient all the time. For example, if user input is cast to an integer, the test case will always be not vulnerable. These missing sanitization functions are found by searching for sanitization functions where all resulting test cases are not vulnerable. If a tool reports any test case as vulnerable, the tool will miss the sanitization functions.

A tool can have fixed results for specific sanitization functions. All sanitization functions that are found in vulnerable and not vulnerable test case are checked for fixed sanitization. It will be checked, if the tool always reports vulnerable or not vulnerable based on a specific sanitization functions. It is important to filter out all test cases that contain unknown sinks or unknown sources beforehand. All the finding can either be that the tool always reports the same vulnerability or it does not report a vulnerability. If no vulnerability is reported (**fixed sanitization**, the tool assumes incorrectly the sanitization functions is always sufficient or the tool cannot continue the data flow analysis from that sanitization function. If the tool always reports a vulnerability for a sanitization function, the tool does not recognize the sanitization function as sufficient in any case (**Missing potential sanitization**).

Table 15.6 shows the results split for XSS and SQLi test cases. It shows that tool A has many *missing sufficient sanitization* functions. Additionally, it shows that tool A has different results on XSS and SQLi test cases. Tool B instead has similar problems on XSS and SQLi test cases. Only one *unknown source* is different. A look into the results show that for that specific source a different vulnerability type is reported (display sensitive data) that is not mapped to XSS. Tool B also has many *missing potential sanitizations*. Overall, both tools have problems with missing sanitization functions that result in false positive reports.

The dataflow test suite has been scanned as well. Because the test suite contains only vulnerable test cases, only false negative and true positive reports can occur. Accordingly, the only useful metric is the recall that calculates how many *dataflow* patterns are supported. Table 15.7 shows the scan results. *Tool A* supported one more *dataflow* pattern than *Tool B*.

15.4.3 Expert interviews

As final evaluation six experts from the Software Assurance Metrics And Tool Evaluation (SAMATE) [218] team have been interviewed about the test suites and the generation process. At first the experts got a presentation about the generation process and a detailed explanation of the decision trees. After the presentation the experts have been interviewed together in an

Table 15.6: Checks on to find missing/problematic patterns.

Checks	**Tool A**	**Tool B**
Unknown sources (XSS)	6	15
Unknown sources (SQLi)	4	14
Unknown sinks (XSS)	3	1
Unknown sinks (SQLi)	7	3
Missing sufficient sanitization (XSS)	30	26
Missing sufficient sanitization (SQLi)	68	26
Fixed sanitization - report (XSS)	3	0
Fixed sanitization - report (SQLi)	2	0
Missing potential sanitization (XSS)	10	39
Missing potential sanitization (SQLi)	11	39

Table 15.7: Scan result on the data flow Test Suite.

Metric	**Tool A**	**Tool B**
False Negative (FN)	12	13
True Positive (TP)	8	7
Dataflow detection (Recall)	40%	35%

online meeting. The interview includes two open ended questions where the experts were free to give their feedback. The first question is related to the distinction between vulnerable and not vulnerable test cases, and the other is related to usefulness of our approach to creating a test suit. The interview was with all experts at the same time, but each expert contributed to each of the questions asked. In the following we will discuss the feedback for each question.

The first question was: *How do you decide between a vulnerable and not vulnerable test case?*
The response from the experts was that the decision for a test case is difficult. It is easy to prove that a test case is vulnerable by providing a working exploit. In contrast, it is hard to prove that a test case is not vulnerable. That is the reason why the definition uses *weakness* to define a vulnerability. A weakness does not mean that it is exploitable. Instead, it is a weak programming construct that might result in an exploitable vulnerability. One expert stated that a good approach to define if a test case is vulnerable or not is to run multiple tools that have been proven to be good. The results can be used to decide if a test case is potentially vulnerable. But it does not ensure that the result is correct. Another expert mentioned that especially sound static code

analysis tools try to solve the issue to distinguishing between false positive and true positive findings.

The second question was: *Are such precise test cases useful as test suite?* One of the experts answered that a large quantity of different test cases is good because it can reveal special cases. The expert mentioned that some static code analysis tools show unpredictable results that are based on the implementation. There can be intended or unintended limitations in the implementation that only trigger on very specific test cases. Additionally, many static code analysis tools use heuristics to decide if a finding is either a false positive or true positive. This can be problematic for synthetic test cases because based on the heuristic a vulnerable test case might not be reported. The heuristics tend to filter out findings on very small code bases because the probability that the vulnerability is artificial is high. The response to the decision tree was positive. The experts did not see any issues that invalidates the approach to use the decision trees to determine between a vulnerable and not vulnerable test case.

To summarize, the response from the experts was positive. The decision tree is a solid way to decide if a test case is vulnerable or not. As they mentioned, the decision for not vulnerable test cases is almost impossible. The experts did not see any issues in the generation process and the decision trees. Accordingly, the decision tree can be used to determine if a test case is vulnerable. The large size of detailed test cases can help to find problems of static code analysis tools. Additionally, the combination of context, sink and filtering to decide if a test case is vulnerable makes it especially interesting for sound static code analysis tools.

15.5 Discussion

We present an approach to decide if a test case is vulnerable or not. A not vulnerable test case cannot be ensured. With the decision tree the likelihood of being vulnerable is very low. Writing an exploit with only numbers and/or alphabetic characters is usually not possible. Such exploits require that the sink does some strange encoding or other functionalities. In contrast, all the test cases that are vulnerable are indeed critical. If such a sample was found in a real project, it would likely be a vulnerability.

Overall the test cases are synthetic. This has advantages and disadvantages. For evaluating static code analysis tools it lets identify where the problem of the tools are similar to the work by Hao [219] - for example, if a vulnerable test case is detected by a tool. Additionally, when another test case with just one different pattern is not detected, that pattern is identified

as a problematic pattern for the tool. The heuristics can be problematic for synthetic test cases. A solution would be to add the test cases in larger projects such that heuristics see the vulnerabilities as unintended.

15.6 Conclusion

Our approach provides a generation process for test cases. Compared to the work from Stivalet and Fong [235], our test cases are an extension of additional patterns. Many different patterns have been used to create a large test suite containing 258,432 test cases that can be used to test the soundness of static code analysis tools and identify problematic patterns. A combination of filtered characters, context and sink allows to decide if a test case is vulnerable or not. The decision tree is a solid way to decide if a test case is vulnerable. Nevertheless, a not vulnerable test case cannot ensure that the test case still might contain a vulnerability. But the likelihood of such a case is very low. The dataflow test suite in contrast is a small test suite, but is important especially for static code analysis tools developers. They can see where problems in the implementation are and what functionality has to be added.

References

[214] Insecurity Refactoring Framework. `https://github.com/fschuckert/insecurity-refactoring`.

[215] NIST. `https://csrc.nist.gov/glossary/term/vulnerability`, .

[216] NIST - Sound Analysis Criteria. `https://www.nist.gov/itl/ssd/software-quality-group/sate-vi-ockham-sound-analysis-criteria`, .

[217] PHP Documentation. `https://www.php.net/manual/`.

[218] SAMATE. `https://www.nist.gov/itl/ssd/software-quality-group/samate`.

[219] Constructing benchmarks for supporting explainable evaluations of static application security testing tools. *Proceedings - 2019 13th International Symposium on Theoretical Aspects of Software Engineering, TASE 2019*, pages 65–72, 2019.

[220] S. Anand. An Orchestrated Survey on Automated Software Test Case Generation. 2013.

[221] T. Boland and P. E. Black. Juliet 1.1 c/c++ and java test suite. *Computer*, 45(10):88–90, oct 2012. ISSN 1558-0814. doi: 10.1109/MC.2012.345.

[222] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[223] O. Charles. Real World software Assurance test Suite: STONESOUP. *In Proceedings of Software Technology Conference*, 2015. URL `{https://conference.usu.edu/stc/Schedule/Grid\_Details.cfm?aid=1955}`.

[224] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. LAVA: Large-Scale Automated Vulnerability Addition. *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, pages 110–121, 2016. doi: 10.1109/SP.2016.15.

[225] K. Goseva-Popstojanova and A. Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, 2015. ISSN 09505849.

[226] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010. doi: 10.1109/TSE.2009.71.

[227] C. Klinger, M. Christakis, and V. Wüstholz. Differentially testing soundness and precision of program analyzers. *ISSTA 2019 - Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 239–250, 2019. doi: 10.1145/3293882.3330553.

[228] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2), Feb. 2011. ISSN 0360-0300. doi: 10.1145/1883612.1883618. URL `https://doi.org/10.1145/1883612.1883618`.

[229] J. Pewny and T. Holz. Evilcoder: automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 214–225, 2016.

[230] F. Schuckert, B. Katt, and H. Langweg. Source Code Patterns of SQL Injection Vulnerabilities. *International Conference on Availability, Reliability and Security*, 2017.

[231] F. Schuckert, M. Hildner, B. Katt, and H. Langweg. Source code patterns of cross site scripting in php open source projects. In *Proceedings of the 11th Norwegian Information Security Conference*, 2018.

[232] F. Schuckert, B. Katt, and H. Langweg. Difficult xss code patterns for static code analysis tools. In *Computer Security*, pages 123–139, Cham, 2020. Springer International Publishing. ISBN 978-3-030-42051-2.

[233] F. Schuckert, B. Katt, and H. Langweg. Difficult sqli code patterns for static code analysis tools. 2020.

[234] F. Schuckert, B. Katt, and H. Langweg. Privacy preservation in federated learning: An insightful survey from the gdpr perspective. *Submitted (Review): Computers & Security*, 2021.

[235] B. Stivalet and E. Fong. Large Scale Generation of Complex and Faulty PHP Test Cases. *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, pages 409–415, 2016. doi: 10.1109/ICST.2016.43.

[236] J. Taneja, Z. Liu, and J. Regehr. Testing static analyses for precision and soundness. *CGO 2020 - Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 81–93, 2020. doi: 10.1145/3368826.3377927.

ISBN 978-82-326-7806-8 (printed ver.)
ISBN 978-82-326-7805-1 (electronic ver.)
ISSN 1503-8181 (printed ver.)
ISSN 2703-8084 (online ver.)

NTNU

Norwegian University of
Science and Technology